DOCUMENT RESUME

ED 318 442 IR 014 352

TITLE	Advanced Placement Computer Science with Pascal. Volume 2. Experimental Edition.
INSTITUTION	New York City Board of Education, Brooklyn, N.Y.
REPORT NO	ISBN-0-88315-689-X
PUB DATE	86
NOTE	341p.; For Volume 1, see ED 288 736.
AVAILABLE FROM	New York City Board of Education, Curriculum Unit, Room 310, P.S. 206, Neck Road and East 22nd Street, Brooklyn, NY 11229.
PUB TYPE	Guides - Classroom Use - Guides (For Teachers) (052)
EDRS PRICE	MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS	*Advanced Placement; College Science; *Computer Science Education; Curriculum Guides; Glossaries; Microcomputers; *Programing; Programing Languages; *Science Activities; Secondary Education; Secondary School Science
IDENTIFIERS	*PASCAL Programing Language

ABSTRACT

This curriculum guide presents 100 lessons for an advanced placement course on programming in Pascal. Some of the topics covered include arrays, sorting, strings, sets, records, computers in society, files, stacks, queues, linked lists, binary trees, searching, hashing, and chaining. Performance objectives, vocabulary, motivation, aim, development, application, summary, and exercises are included for each lesson. The appendix includes instructions for using Turbo Pascal. A glossary of computer terms and a questionnaire for evaluating this guide are attached. (MES)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED318442

NEW YORK CITY BOARD OF EDUCATION

DIVISION OF CURRICULUM AND INSTRUCTION
AND THE OFFICE OF TECHNOLOGY

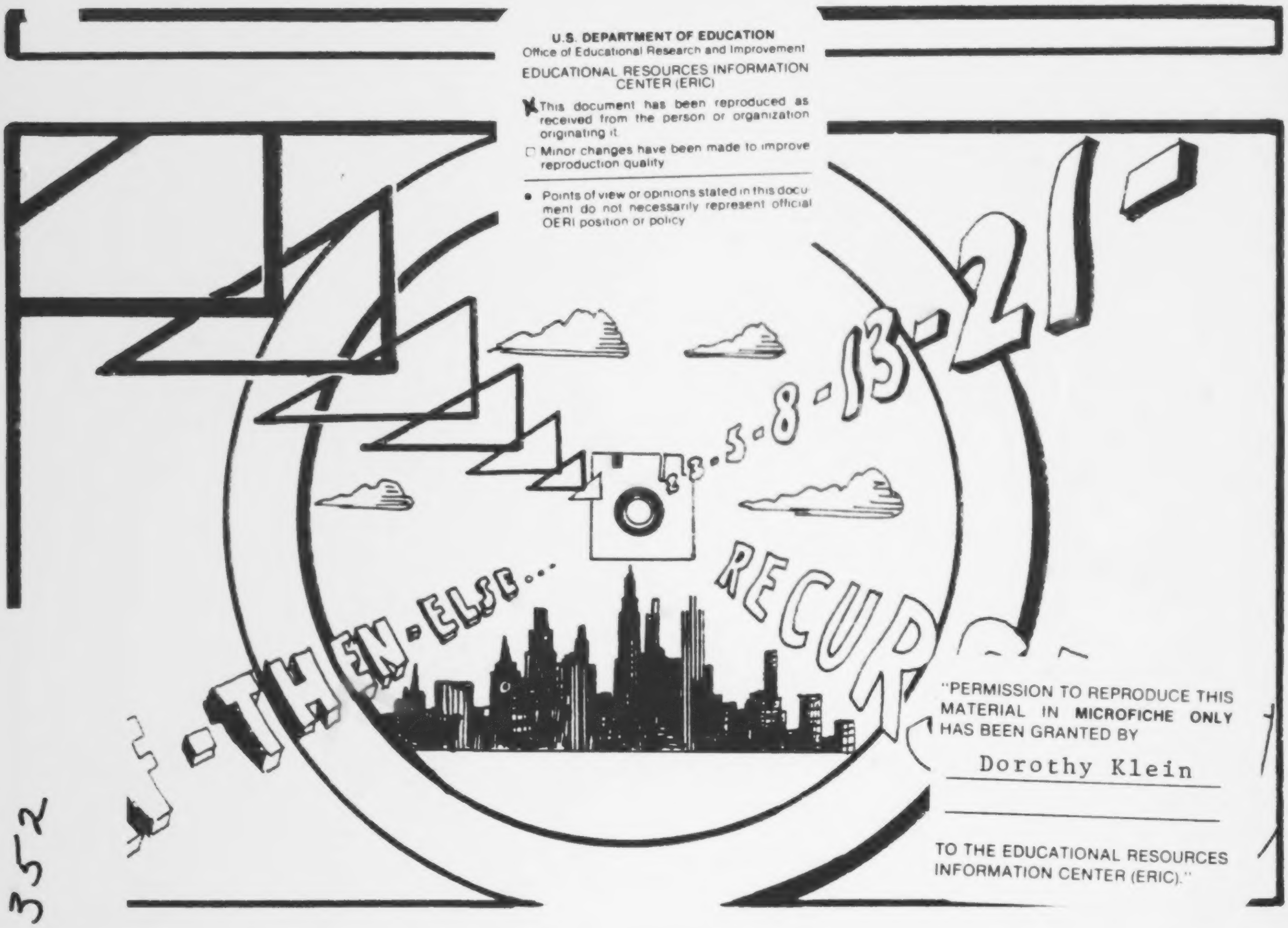
ADVANCED PLACEMENT COMPUTER SCIENCE WITH PASCAL

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☒ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality

- Points of view or opinions stated in this document do not necessarily represent official OERI position or policy



"PERMISSION TO REPRODUCE THIS MATERIAL IN **MICROFICHE ONLY** HAS BEEN GRANTED BY

Dorothy Klein

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

IR014352

VOLUME 2

EXPERIMENTAL

BEST COPY AVAILABLE

New York City Board of Education Teacher's Guide to

ADVANCED PLACEMENT
Computer Science
(with Pascal)

VOLUME 2 - EXPERIMENTAL EDITION

New York City Board of Education

James F. Regan, *President*
Miguel O. Martinez, *Vice President*

Joseph G. Barkan
Stephen R. Franse
Irene Impellizzeri
Marjorie A. Lewis
Robert F. Wagner, Jr.

Nathan Quinones, *Chancellor*

Charles I. Schonhaut, *Deputy Chancellor*

Jerald Posman, *Deputy Chancellor for Financial Affairs*

Louise Latty, *Chief Executive for Instruction*

Charlotte Frank, *Executive Director*
Division of Curriculum and Instruction

Irwin Kaufman, *Director*
Office of Technology

Copyright (c) 1986
by the Board of Education of the City of New York

Application for permission to reprint any section of this material should be made to the Chancellor, 110 Livingston Street, Brooklyn, N.Y. 11201. Reprint of any section of this material shall carry the line, "Reprinted from *Advanced Placement Computer Science With Pascal* by permission of the Board of Education of the City of New York."

New York City Public Schools should order additional copies of this publication from the Bureau of Supplies.

ISBN No. 0-88315-689-X Curric. No. 00-7000-41

Other persons and institutions may obtain copies of this publication from the Curriculum Production Unit of the Office of Curriculum Development and Support, Room 613, 131 Livingston Street, Brooklyn, N.Y. 11201 For information call (718) 596-4902.

NEW YORK CITY BOARD OF EDUCATION
Division of Curriculum and Instruction
Office of Curriculum Development and Support

New York City Board of Education Teacher's Guide to

ADVANCED PLACEMENT
Computer Science
(with Pascal)

VOLUME 2 - EXPERIMENTAL EDITION

NOTES ON THE USE OF THIS GUIDE

This book is the second part of a two volume series. The series was designed to provide teachers with enough support material to enable them to teach an advanced course in Pascal programming. It is intended for teachers whose classes have completed all of the work presented in our earlier programming guides. Volume II extends the objectives of Volume I.

The topics in these books are each designed to encompass an entire unit of instruction rather than a specific time period. Some may require only a few minutes to present while others will take a few days to complete.

The average high school class studying this material for three lecture and two lab periods per week, should be able to successfully complete all of the topics in approximately three semesters.

Since each classroom situation is different, it is the staff of each school that must determine how this guide can be used most effectively. In fact the individual topics may be used in their current format or may be adapted to meet the particular needs of various groups of students.

FOREWORD

In 1983 the Division of Curriculum and Instruction published computer literacy curriculum guides for both elementary and secondary schools. That project was the first step in a comprehensive plan for computer education in New York City. Computer technology is new and it is changing daily. We as educators are faced with a common challenge: how to use this exciting and powerful technology in a way that will meet the changing needs of the students and will continue to accommodate changes as they occur. The challenge is a great one, but the potential for these tools is even greater.

We know that computers can ignite the imagination and extend the limits of human learning. They enable their users to view the world in new ways. When students interact with computers they take charge of their own learning and learn more through this interaction. In short, computers are tools used by humans for human endeavors.

New York City Board of Education Teachers's Guide to Advanced Placement Computer Science (with Pascal) is an important step in the New York Cit plan for computer education. Pascal is a sophisticated and flexible computer language. As students gain proficiency in Pascal, they develop important concepts in computer science. The content of this advanced placement curriculum will challenge students to master the reasoning skills demanded for programming Pascal. We hope that this guide will aid teachers and students in keeping pace with the challenges of our information age.

Charlotte Frank

Charlotte Frank
Executive Director
Division of Curriculum and Instruction

ACKNOWLEDGMENTS

New York City Board of Education Teacher's Guide to Advanced Placement Computer Science with Pascal, Volume 2 (Experimental Edition) is a joint project of the Division of Curriculum and Instruction, Charlotte Frank, Executive Director, and the Office of Technology, Irwin Kaufman, Director.

Marc Licht Director and Joseph Eaione Assistant Director of the Computer and Information Sciences Unit gave impetus and guidance to this project by supervising the writing teams, arranging for the planning and production meetings, and screening materials produced by the writers.

Stephen Radin, Director of Curriculum Development served as project coordinator of the second volume and also as an author.

A team of dedicated professionals wrote the topics and were involved in all aspects of their compilation:

Yvette Beck, Herbert H. Lehman H.S.
Alice Farkouh, Curtis H.S.
Mary Gahn, John Bowne H.S.
Judy Kotkin, Brooklyn Technical H.S.
Art Seelenfreund, Herbert H. Lehman H.S.
Nick Skouras, John Bowne H.S.
Dave Weinstein, Brooklyn Technical H.S.

The following reviewers made helpful comments and suggestions: Phil Bayman, Christopher Columbus H.S.; James DeLuca, Hillcrest H.S.; Neal Ehrenberg, Computer and Information Sciences Unit; Joel Friedberg, Herbert Lehman H.S.; Melvin Klein, Brooklyn Technical H.S.; Judy Kotkin, Brooklyn Technical H.S.; Marc Licht, Computer and Information Sciences Unit; Ellen Norden, Bronx H.S. of Science; Richard Rothenberg, Stuyvesant H.S.; Mark Saul, Bronx H.S. of Science; Elayne Schulman, Staten Island Academy; Arnold Scheiman, Park West H.S.; Nick Skouras, John Bowne H.S. and Dave Weinstein, Brooklyn Technical H.S.

The manuscript was prepared for publication by the Computer and Information Sciences Unit, under the direction of Stephen Radin.

Editing:	Dave Weinstein, Nick Skouras
Word processing:	Janet Taylor
Graphics:	Ernest Mortuzans
Cover:	Victor Roseman

INTRODUCTION

The use of computers in the classroom has become more prevalent during the past few years. Through this use we have developed a knowledge of their functions and a better vision of their potential. We encourage our colleagues to freely experiment and attempt new approaches. Only with all of us working in concert, can we fulfill our vision for an effective integration of school technology into all subject areas. From these beginnings, we will nurture the critical thinking and practical skills our students will need to succeed in the information age. If children are not comfortable with computers, they will as be unprepared for the future as an adult who cannot read or write.

The purpose of this guide is to give teachers and supervisors a working knowledge of various approaches to enhance pupil learning. Our guides are intended tp be open ended and flexible in nature. As a 1920 car is not efficient by 1980 standards, outmoded computer education ideas of the seventies have been replaced by the new visions of the eighties. Classroom use of technology has doubled and redoubled in the past few years. Teachers are continually devising new ideas for the use of this technology.

Our primary responsibility in this area and our reason for writing this book, is to see that computers are used as an efficient classroom tool. Only through a real sharing can we move forward in this rapidly changing field of technology. Feel free to write to me directly with your comments and suggestions concerning this curriculum guide, so that we may improve and update it to meet your needs.

Irwin Kaufman

Irwin Kaufman
Director
Office of Technology

It is the policy of the New York City Board of Education not to discriminate on the basis of race, creed, national origin, age, handicapping condition, or sex in its educational programs, activities, and employment policies, as required by law. Inquiries regarding compliance with appropriate laws may be directed to Mercedes Nesfield, Director, Office of Equal Opportunity, 110 Livingston Street, Brooklyn, N.Y. 11201; or to Charles Tejeda, Director, Office of Civil Rights, Department of Education, 26 Federal Plaza, Room 33-130, New York, New York 10278

TABLE OF CONTENTS

TOPIC 39 - ARRAYS, A STRUCTURED DATA TYPE 1

TOPIC 40 - USING ARRAYS IN PROGRAMS 5

TOPIC 41 - ARRAYS AS PARAMETERS IN PROCEDURES 8

TOPIC 42 - SORTING 12

TOPIC 43 - BINARY SEARCH 16

TOPIC 44 - IMPLEMENTING MULTI-DIMENSIONAL ARRAYS 24

TOPIC 45 - PACKING ARRAY ELEMENTS TO REDUCE STORAGE 30

TOPIC 46 - IMPLEMENTING STRINGS WITH PACKED ARRAYS 35

TOPIC 47 - IMPLEMENTING STRINGS WITH UCSD PASCAL 40

TOPIC 48 - UCSD STRING PROCESSING APPLICATIONS 44

TOPIC 49 - STANDARD PASCAL STRING PROCESSING 49

TOPIC 50 - SETS IN PASCAL AND THEIR OPERATIONS 57

TOPIC 51 - ILLUSTRATION OF SET PROPERTIES 62

TOPIC 52 - PASCAL RECORDS 66

TOPIC 53 - ACCESSING INFORMATION STORED IN RECORDS 70

TOPIC 54 - ACCESSING DATA STORED IN HIERARCHICAL RECORDS 75

TOPIC 55 - ACCESSING DATA STORED IN VARIANT RECORDS 80

TOPIC 56 - COMPUTERS IN SOCIETY 84

TOPIC 57 - INTRODUCTION TO FILES 89

TOPIC 58 - WRITING FILES 94

TOPIC 59 - READING FILES 99

TOPIC 60 - COMBINING FILES 104

TOPIC 61 - UPDATING FILES 108

TOPIC 62 - TEXTFILES 114

TOPIC 63 - TEXTFILE APPLICATIONS 119

TOPIC 64 - STACKS -- A DYNAMIC DATA STRUCTURE 124

11

TOPIC 65 - IMPLEMENTING A STACK 128

TOPIC 66 - USING A STACK FOR ERROR CHECKING 133

TOPIC 67 - POSTFIX EXPRESSIONS 137

TOPIC 68 - THE STRUCTURE OF A QUEUE 142

TOPIC 69 - IMPLEMENTATION OF QUEUE OPERATIONS 146

TOPIC 70 - LINEAR LINKED LISTS 151

TOPIC 71 - STACKS AND QUEUES USING LINKED LISTS 159

TOPIC 72 - PASCAL IMPLEMENTATION OF LINKED LISTS 163

TOPIC 73 - LINKED LIST OPERATIONS IN PASCAL 169

TOPIC 74 - CIRCULAR LINKED LIST 174

TOPIC 75 - HEADER NODES AND DOUBLY LINKED LIST 178

TOPIC 76 - POINTER VARIABLES 182

TOPIC 77 - DYNAMIC LINKED LIST 186

TOPIC 78 - LINKED LIST OPERATIONS 189

TOPIC 79-80 - AIRLINE RESERVATION SYSTEMS 194

TOPIC 81 - BINARY TREES 202

TOPIC 82 - BINARY TREE SORT 207

TOPIC 83 - BINARY TREE TRAVERSAL 213

TOPIC 84 - ARRAY IMPLEMENTATION OF BINARY TREES 218

TOPIC 85 - DYNAMIC IMPLEMENTATION OF BINARY TREES 224

TOPIC 86 - HUFFMAN ALGORITHM 228

TOPIC 87 - NON-BINARY TREES 237

TOPIC 88 - TREE REPRESENTATION OF GENERAL EXPRESSIONS 243

TOPIC 89 - INTRODUCTION TO SORTING 249

TOPIC 90 - SORTING -- THE BUBBLE SORT 253

TOPIC 91 - BINARY TREE SORT 257

TOPIC 92 - SELECTION SORT 262

TOPIC 93 - INSERTION SORT 267

TOPIC 94 - MERGE SORT 273

TOPIC 95 - SEARCHING OVERVIEW AND SEQUENTIAL SEARCH 277

TOPIC 96 - EFFICIENCY OF SEARCH ALGORITHMS 281

TOPIC 97 - INDEXING A TABLE 284

TOPIC 98 - SEARCHING A BINARY TREE 289

TOPIC 99 - HASHING 294

TOPIC 100 - CHAINING 298

APPENDIX

USING TURBO PASCAL 303

GLOSSARY 313

QUESTIONNAIRE 329

Advanced Placement Computer Science With Pascal

TOPIC 39

ARRAYS, A STRUCTURED DATA TYPE

Performance Objectives - Students will be able to:

- explain why structured types are needed
- define an array and related terms
- write an array type declaration

Vocabulary: structured type, array, record, file, set, subscript, index, array component (element), array type declaration, subrange, ordinal type.

Motivation:

- Write a routine to read in a list of ten values and store them in memory.
  Answer: Read (A, B, C, D, E, F, G, H, I, J);

- How would we store 50 or 100 such values?

- What are four simple (unstructured) data types? (boolean, char, real and integer)

Aim: How do we create a structured data type that will represent a collection of values?

Development:

1. Pascal allows for the structuring of variables to accommodate many programming situations. A structured type is composed of simple data types. A simple variable can only hold one type.

   - Pascal has four built-in structured types. They are set, array, record and file.

2. The set is a data structure that stores a collection of values and provides an easy means to compare different sets and test whether a value is in that set.

```
TYPE
  letterset = SET OF 'A'..'F';
```

3. The file is a data structure that allows permanent storage of a collection of data. We may read values from a file and also write values to a file.

```
TYPE
  numbers = FILE OF integer;
```

4. The record structure stores components that may be of different data types in fields. Each field may be individually accessed.

```
TYPE
  studentdata =
    RECORD
      idnumber,
      coursecode : integer;
      grade : char;
      passed : boolean
    END;
```

5. The array structure can hold any number of values of the same type. The elements of an array are subscripted variables. By specifying the subscript or index of an array identifier, we can access the contents of its storage location.

```
TYPE
  table = ARRAY [1..100] OF real;
```

- If we have a large collection of values of the same type that must be stored, then the structure to use is the array.

6. Let us analyze the creation of an array type.

```
TYPE
  table = ARRAY [1..100] OF real;
```

- Every array can be given a type identifier. "Table" is the type indentifier of this array. The values inside the square brackets are the values used to access the 100 different locations of this array. These values are called the subscripts or indices of the array. Note that the bounds of the array, 1 and 100, are separated by two periods.

- How would you declare "letter" to be an array type which will store up to 100 characters?

```
TYPE
  letter = ARRAY [1..100] OF char;
```

7. The bounds of an array can be any subrange of an ordinal type. Therefore, the following type declarations are valid.

```
TYPE
  list = ARRAY [-5..5] OF integer;
  seat = ARRAY ['A'..'F'] OF char;
```

- Since user defined types are ordinal, they may also serve as bounds for an array. In addition, an ordinal type identifier itself (except integer) may be used to set the bounds of an array.

- Declare type "days" to be the five weekdays, then declare "hours" to be an array whose subscripts are those five days of the week.

```
TYPE
  days = (Mon, Tues, Wed, Thurs, Fri);
  hours = ARRAY [days] OF real;
```

- Note that each array definition uses two data types. The bounds which serve as indices for the array have their own ordinal type. The individual components of the array may be of another type specified following the word "OF".

8. Once these array types have been defined, we may declare variables to be of this type.

   - Define "list" to be an array of four integers. Then declare "A" and "B" to be variables of this type.

```
TYPE
  list = ARRAY [1..4] OF integer;
VAR
  A, B : list;
```

   - Let's look at a schematic of what variable A might look like in memory.

	A[1]	A[2]	A[3]	A[4]
A	5	53	25	1

   - A memory block "A' has been set aside consisting of four memory locations. In this example, location A[3] stores the element 25.

   - When coding a program to refer to specific locations of an array, we enclose the index in square brackets.

     - For example, the above values could have been placed in the array by the assignments:

```
A[1] := 5;
A[2] := 53;
A[3] := 25;
A[4] := 1;
```

   - Write a Pascal command to print the element stored in the cell whose subscript is 4.

     - writeln (A[4]);

9. The bounds for an array type <u>may not</u> be variables, because variable definitions follow type definitions. However, bounds may be declared constants. Subscripts for arrays, in a program, may be variables.

   - For example, the following declaration is illegal <u>unless</u> X was predefined as a constant.

```
TYPE
  list = ARRAY [1..X] OF real;
```

   - The assignment: `Y := A[2*X+1]` would be valid, assuming X and Y were predeclared variables, and there was no type mismatch.

Summary:

1. What is a structured data type?
2. What are four structured types used in Pascal?
3. What is an array?
4. What must be done before an array variable can be used?
5. How are the bounds of an array defined?
6. How do we give values to and extract values from the components of the array?

Exercises:

1. Declare "month" to be an array type to store real values for the 31 days of the month:

```
TYPE
  month = ARRAY [1..31] OF real;
```

2. Declare "January" to be a variable of this type

```
VAR
  January : month;
```

3. Assign a temperature of 36.5 to the location that would represent January 3rd.

```
January[3] := 36.5;
```

TOPIC 40

USING ARRAYS IN PROGRAMS

Performance Objectives - Students will be able to:

- use arrays to solve programming problems
- explain when arrays are used in problem solving

Motivation:

- Write a program to input a list of 100 integers.

```
PROGRAM Donow (input);
TYPE
  list = ARRAY [1..100] OF integer;
VAR
  X : integer;
  L : list;
BEGIN
  FOR X := 1 TO 100 DO
    readln (L[X])
END.
```

Aim: How to use arrays to help us solve programming problems.

Development:

1. In the program, array L contains 100 integers which may be accessed individually, compared, and changed.

2. Write a program segment to multiply all negative elements of array L by -1. Assume "counter" has been declared to be of type integer.

```
FOR counter := 1 TO 100 DO
  IF L[counter]< 0 THEN
    L[counter] := L[counter] * (-1);
```

3. The index of an array may indicate more than mere positional notation.

   - Suppose the I.D. numbers for students in a school ranged from 1000 to 4999. Define an array so that a student's average may be accessed by means of his I.D. number.

```
TYPE
  ID = ARRAY [1000..4999] OF real;
```

4. Arrays should be considered when a problem involves operations on a fixed number of variables of the same type. When the number of variables is large and computations require all the previous values later on in the program, an array will probably be essential.

5. A frequent application of an array is to search the list for maximum or minimum values. If array "temp" stores the hourly temperatures in a 24-hour period, write a routine to search the array for the high and low temperature readings.

   . First create the variable declaration block. In this example, the array variable will be declared without using a type definition.

```
VAR
  hi, lo : integer;
  temp : ARRAY [1..24] OF integer;
```

   . Then read the temperature values into the array:

```
FOR I := 1 TO 24 DO
  readln (temp[I]);
```

   . Next set the hi and lo temperatures equal to the first entry in the array. As we scan the list, change their values accordingly.

```
lo := temp[1];
hi := temp[1];
FOR I := 2 TO 24 DO
  IF temp[I] > hi THEN
    hi := temp[I]
  ELSE
    IF temp[I] < lo THEN
      lo := temp[I];
```

6. We should be able to display the contents of an array in a specified format. Write a routine to send the contents of the array above to the screen displaying four temperatures per line.

```
FOR I := 1 TO 24 DO
  BEGIN
    write (temp[I] : 5);
    IF I MOD 4 = 0 THEN
      writeln
  END;
```

7. Can each element of an array be a string of characters?

   . Standard Pascal does not incorporate strings as a structured data type. However, we can mimic string representation by defining each element of an array to be an array of characters.

   . Later we will use packed arrays of characters to facilitate this representation.

8. Create a data type "word" to be a structure that will store a 5-letter word. Write a program that asks the user to enter a 5-letter word and then stores it into the data structure. Finally, output this array of five characters as a single word.

```
PROGRAM Aword (input, output);
TYPE
  word = ARRAY [1..5] OF char;
VAR
  word1 : word;
  X : integer;
BEGIN
  writeln ('Enter a 5-letter word: ');
  FOR X := 1 TO 5 DO
    read (word1[X]);
  writeln;
  FOR X := 1 TO 5 DO (* echo the word to the screen *)
    write (word1[X])
END. (* Aword *)
```

Summary:

1. When are arrays used in problem solving?

Exercises:

1. Write a routine to compute the average of an array of 30 integers.
2. "Averages" is an array of all student averages in a class of 35. Write a routine to print all averages that are 65 or above together with their subscripts.

TOPIC 41

ARRAYS AS PARAMETERS IN PROCEDURES

Performance Objectives - Students will be able to:

- explain how to manipulate arrays when the exact number of elements is determined during execution
- write procedures using arrays as parameters
- indicate why an array is a random access structure

Vocabulary: random access structure

Motivation:

- Write a type declaration for "roster" to be an array of 25 integers. The roster will store the player jersey numbers for each of the 25 players on a baseball team. Declare Yankees, Tigers, Redsox, and Orioles to be array variables of this type.

```
TYPE
  roster = ARRAY [1..25] OF integer;
VAR
  Yankees, Tigers, Redsox, Orioles : roster;
```

- How could we read in the player jersey numbers for each team?
  - We could use a FOR/DO loop four times to read in the data for each team.
  - We could write a procedure and call it for each team.
- What problem does the second method pose? (We would have to pass an array as a parameter to the procedure.)

Aim: How do we pass array parameters to a procedure?

Development:

1. The method is essentially the same as with the simple data type parameters. To pass the array "Yankees" to the procedure "Bringin", we simply call: Bringin (Yankees) from the main program.
   - Since we want any changes to Yankees to be reflected in the main program, we pass Yankees to a variable parameter in the procedure.
   - Also, by not using value parameters, local copies of the array are not made. We save memory and computer time.
2. Write procedure Bringin using "team" as a variable parameter to read in the jersey numbers of all 25 players on the team.

```
PROCEDURE Bringin (VAR team : roster);
VAR
  I : integer;
BEGIN
  FOR I := 1 TO 25 DO
    BEGIN
      writeln;
      write ('Enter jersey number of player', I);
      read (team[I])
    END
END;
```

- When using an array as a parameter, you must use a previously declared identifier as the array type, such as "roster" (in the above example), rather than the usual "ARRAY [1..25] OF integer" declaration.

- When Bringin (Yankees) is called from the main program, Yankees replaces team and since we're using a call by reference, the values assigned to team are actually assigned to Yankees back in the main program.

- Suppose we want to print out the jersey number of player 18 from the main program. Do we have to scan through all the players to access information about the 18th player? No -- since in the array we can print the 18th element directly by using the statement: write (Yankees[18]). A structure which allows you to access any element directly is called a random access structure.

- By using the subrange 1..25 in our example, we assume each team has an active roster of the maximum 25 players allowed. Can we instead read in the actual roster limit for each team, and dimension our array accordingly? No -- the type declaration of arrays precedes the main program.

3. Assume a team has a roster less than 25 players. What problem does this pose? (The FOR/DO loop attempts to read in 25 values.) How can we manipulate the array, so the exact number of elements is not to be determined until execution time?

   - Determine the maximum number of elements the array might have. For our example we assume 25 is the legal roster limit (max = 25). We use the index "count" to keep track of the number of players read into the roster, as well as the particular player being accessed. If the team has fewer than 25 players, the user indicates when he has entered the last player by typing -1 as a sentinel. In that way, the read loop continues until (team[count] = sentinel) OR (count = max).

4. Rewrite procedure Bringin to include variable parameters for "count" and "team" and value parameters for "max" and "sentinel." Use a WHILE/DO loop to control the input data.

```
PROCEDURE Bringin (VAR team : roster; max, sentinel : integer; VAR
                  count : integer);
VAR
  temp : integer;
BEGIN
  write ('Enter player number, or the sentinel -1 to end: ');
  count := 0;
  readln (temp);
  WHILE (count < max) AND (temp<>sentinel) DO
    BEGIN
      count := count + 1;
      team[count] := temp;
      readln (temp)
    END
END;
```

. To call this procedure from the main program we can use:
  Bringin (Yankees, 25, -1, ycount);

5. Write a routine for the main program that prints all the player jersey numbers for the Yankees.

```
writeln ('Player' : 10, 'Number' : 10);
FOR I := 1 TO ycount DO
  writeln (I : 10, Yankees[I] : 10);
```

. Since "ycount" is passed to the variable parameter "count", its value equals the current roster of the Yankees which may be less than the dimensioned value of the array.

Summary:

1. When is it useful to use a procedure call? (If several lists are to be modified the same way.)

2. What effect does using variable parameters for the arrays have on the main program? (Any modifications made on the variable parameters are reflected in the main program.)

3. How are array elements accessed? (Since the array is a random access structure, the elements are accessed directly.)

Exercises:

1. Where A, B, C, X, Y and Z are arrays of integers, describe the array C that is modified by the procedure call: Addum (A, B, 7, C);

```
PROCEDURE Addum (X, Y : numberarray; I : integer; VAR Z : numberarray)·
VAR
  J : integer;
BEGIN
  FOR J := 1 TO I DO
    Z[J] := X[J] + Y[J]
END;
```

- Solution: Array C is the element by element sum of the first seven elements of arrays A and B.

- Discuss the pros and cons of defining Z as a variable parameter and X and Y as value parameters.

2. Write a procedure that determines whether a particular value is in a specified array.

TOPIC 42

SORTING

Performance Objectives - Students will be able to:

. discuss the importance of sorting arrays
. describe bubble sort techniques
. write a bubble sort procedure

Vocabulary: bubble sort

Motivation:

. Write a program to input an array of 10 integers.

```
PROGRAM Numbers (input);
VAR
  list : ARRAY [1..10] OF integer;
  X : integer;
BEGIN
  FOR X := 1 TO 10 DO
    readln (list[X])
END.
```

. How can we determine whether specified values are located in the array?

- One method is to check the items in the array one by one. Search efficiency can often be improved if the items in the array are first sorted.

Aim. How can we sort the elements of an array?

Development:

1. Numbers can be sorted in increasing or decreasing order. Lists of words can be sorted alphabetically.

2. Scores of sorting routines have been developed. An efficient routine is marked by fewer comparisons and the minimization of computer time and space. Efficiency considerations become significant as the volume of data to be sorted increases.

3. A popular sorting technique is the bubble sort. Suppose we view the N items in an array to be in linear order from left to right. The bubble sort compares the first two items in the array and switches them, if necessary, so that the larger item is on the right. The sort continues to compare adjacent items in order (2nd to 3rd, 3rd to 4th, and so on) until the largest value has been positioned on the rightmost side of the array (bubbled to the right).

- After this first pass, the sort repeats the process on the remaining N-1 leftmost items until the second largest value in the array has bubbled to the second position from the right. The sort continues to make these comparisons N-1 times, each time with one less item, until all the items in the array have been sorted.

4. Given the list of numbers 22 17 25 20 12 examine the first pass with a bubble sort:

```
  ^
22  17  25  20  12     Switch (to get larger number on right)
      ^
17  22  25  20  12     No switch
          ^
17  22  25  20  12     Switch
              ^
17  22  20  25  12     Switch

17  22  20  12  25   (The largest number "25" has bubbled to the right.)
```

- The following are the positions of the numbers after
  - pass two: 17 20 12 22 25
  - pass three: 17 12 20 22 25
  - pass four: 12 17 20 22 25
- Note that there are four passes to get the five items in order. Each pass involved comparisons with one less item than the previous pass.

5. The bubble sort on N items can be implemented using a nested FOR loop:

```
FOR I := 1 to N-1 DO
  FOR J := 1 to N-I DO
    IF A[J] > A[J+1] THEN
      Switch (A[J], A[J+1]);
```

- Have the students create the Switch procedure.

```
PROCEDURE Switch (VAR N1, N2 : integer);
VAR
  temp : integer;
BEGIN
  temp := N1;
  N1 := N2;
  N2 := temp
END; (* Switch *)
```

6. How many comparisons were needed to sort five items? (4 + 3 + 2 + 1 = 10) How many comparisons would be needed to sort 50 items? (49 + 48 + 47 + ... + 1 = 1225) Those comparisons would be done even if the list

(unknowingly) was originally ordered or became ordered after a few passes.

How can we rewrite the bubble sort routine so that we stop the comparisons as soon as the list is ordered?

Write a procedure called Bubblesort, using a REPEAT-UNTIL structure:

```
REPEAT
  compare and switch process
UNTIL
  there are no switches
```

- We will need two local variables in our procedure.
  - One to determine whether another pass is necessary -- a boolean value to see if a switch has been made.
  - A second to determine the number of items we need to compare on the next pass -- one less than on the preceding pass.
- We can set boolean identifier ASWITCH := FALSE initially and reassign it a value of "true" when a switch has been made -- indicating the necessity of another pass.

7. Here is the routine:

```
PROCEDURE Bubblesort  (VAR list : numbersarray; items : integer);
      (* numbersarray is a globally defined array type of reals *)
      (* items is the number of elements in the array list *)
VAR
  aswitch : boolean;
  I : integer;
BEGIN
  REPEAT
    aswitch:= false; (* initially no switches have been made *)
    items := items - 1;
    FOR I := 1 TO items DO
      IF list[I] > list[I + 1] THEN
        BEGIN
          switch (list[I], list[I +1]);
          aswitch := true
        END
  UNTIL NOT aswitch
END;
```

Summary:

1. Define Sorting. (The process of rearranging values so that they satisfy a particular order relationship.)

2. What is a bubble sort?

3. What efficiency considerations should one take into account in writing a sorting routine?

Exercises:

1. Given the list: 10 4 12 9

   - Give the order after each bubble sort pass.

   - How many passes would be necessary to order the list using a bubble sort routine which did not include the Boolean variable aswitch? (3) which did include aswitch? (2)

2. Write a program to input a list of 15 real numbers. Sort the list in decreasing order. Print the original and sorted lists in adjacent columns.

3. Rewrite the bubble sort using a WHILE structure.

TOPIC 43

BINARY SEARCH

Performance Objectives - Students will be able to:

- state the need for efficient search techniques
- identify problems related to a sequential search
- explain how the binary search method reduces search time
- write a recursive binary search function

Vocabulary: binary search

Motivation:

- One hundred students are enrolled in a school. Each is assigned a 4-digit I.D. number. Write a program that determines whether a student with an I.D. number of 4325 is on the list of enrolled students.

- Prior to attempting the program, what data structure is suggested by the word "list"? (An array)

- How can the I.D. numbers or elements be stored in the array?

  - in numerical order (ascending)
  - in any order (haphazard)

- To determine whether a specific element is in the list, what type of technique must be used? (a search)

Aim: How can an array be searched for a given value?

Development:

1. There are two simple types of searching techniques: sequential (linear) search and binary search.

2. A sequential search is a technique in which each element of an array is examined in succession and then compared to the desired element. This process continues until a match is found.

   - When a list is unordered, a sequential search is necessary to find a desired item.

   - A sequential search may be used to find an element in an ordered list, but it is not the most efficient method. Searching each element of an array is very time-consuming.

     - For an ordered list, a binary search is more efficient and less time-consuming.

3. The binary search algorithm uses the principle of continuously dividing a list in half and determining whether the desired number is in the upper or lower half of the list. The algorithm is as follows:

   - Determine the middle of the list.
   - Compare the value that you are searching for to the element in the middle of the list.
   - If they are equal, the search is successful.
   - If the value is less than the element in the middle of the list, the value is in the bottom half of the list. Redefine the list so that the element in the middle less one becomes the top of the new list.
   - If the value is greater than the element at the middle of the list, then the value is in the upper half of the list. Redefine the list so that the element in the middle plus one becomes the bottom of the new list.
   - The process repeats itself until the value is found or the search proves negative.

4. Simulate the above process with the following list of 10 elements. (The list is an ordered array.)

   10 15 18 25 26 34 39 42 41 44

   - Search for the element equal to 18.

	bot				mid					top
	↓				↓					↓
position	1	2	3	4	5	6	7	8	9	10
list	10	15	18	25	26	34	39	42	41	44

   - First Probe

     - The first step requires finding the middle element:

       mid = (bot + top) div 2
       = (1 + 10) div 2
       = 11 div 2
       mid = 5

     - The next few steps determine whether the value being sought (18) is =, <, or > the element in the middle of the list and redefines the list if necessary.

       Is 18 = list[mid]? (list[5])
       18 < list[mid]

     - The element in list[mid] (list[5]) is 26. Since the value we are looking for is less than the middle value, the top half of the list is eliminated. The top of the new list is defined to be mid - 1.

- <u>Second Probe</u>

  - The list is now redefined with four elements.

```
                bot          top
                 ↓            ↓
    position     1    2    3    4
    list       | 10 | 15 | 18 | 25 |
```

```
    mid = (bot + top) div 2
        = (1 + 4) div 2
        = 5 div 2
    mid = 2

    Is 18 = list[mid]?
    18 > 15
```

  - Since 18 > list[2], the bottom part of the list is eliminated and the new bottom is reset to be mid + 1.

- <u>Third Probe</u>

  - Redefine list.

```
                bot  top
                 ↓    ↓
    position     3    4
    list       | 18 | 25 |
```

```
    mid = (bot + top) div 2
        = (3 + 4) div 2
        = 7 div 2
    mid = 3


    Is 18 = list[mid]?
    18 = 18
```

  - The desired value was found using three probes.

5. Repeat the above binary search. This time locate the value 44.

```
             bot                                            top
              ↓                                              ↓
position      1    2    3    4    5    6    7    8    9    10
list        | 10 | 15 | 18 | 25 | 26 | 34 | 39 | 41 | 42 | 44 |
```

- <u>First Probe</u>

```
    mid = (bot + top) div 2
        = (1 + 10) div 2
        = 11 div 2
    mid = 5
```

Is 44 = list[5]?
44 > 26

- Since the middle value is less than the number we are searching for, the bottom half of the list is eliminated. In the second probe, we must search from the mid + 1 position of the array to the top.

- We can now redefine our list.

- **Second Probe**

	bot ↓				top ↓
position	6	7	8	9	10
list	34	39	41	42	44

```
mid = (bot + top) div 2
    = (6 + 10) div 2
    = 16 div 2
mid = 8
```

Is 44 = list[mid]?
44 > 41

- **Third Probe**

  - Redefine list from mid + 1 to top.

	bot ↓	top ↓
position	9	10
list	42	44

```
mid = (bot + top) div 2
    = (9 + 10) div 2
    = 19 div 2
mid = 9
```

Is 44 = list[mid]?
44 > 42

- **Fourth Probe**

  - Redefine list.

	bot ↘ ↙ top
position	10
list	44

```
mid = (bot + top) div 2
    = (10 + 10) div 2
    = 20 div 2
mid = 10
```

Is 44 = list[mid]?
44 = 44

- How many probes did it take to find the desired value? (4)

6. Repeat the binary search. Try to locate the value 19.

. <u>First Probe</u>

	bot ↓									top ↓
position	1	2	3	4	5	6	7	8	9	10
list	10	15	18	25	26	34	39	41	42	44

```
mid = (bot + top) div 2
    = (1 + 10) div 2
mid = 5
```

Is 19 = list[mid]?
19 < 26

. <u>Second Probe</u>

- Redefine the list.

	bot ↓			top ↓
position	1	2	3	4
list	10	15	18	25

```
mid = (bot + top) div 2
    = (1 + 4) div 2
mid = 2
```

Is 19 = list[mid]?
19 > 15

. <u>Third Probe</u>

- Redefine the list.

	bot ↓	top ↓
position	3	4
list	18	25

```
mid = (bot + top) div 2
    = (3 + 4) div 2
mid = 3
```

Is 19 = list[mid]?
19 > 18

- **Fourth Probe**

  - Redefine the list.

```
                 bot  top
                   ↘ ↙
    position        4
    list          [25]

    mid = (bot + top) div 2
        = (3 + 3) div 2
    mid = 3

    Is 19 = list[mid]?
    19 < 25
```

  - When the list is redefined, the top is set to 3 (mid - 1 = 3) and the bottom remains 4, thereby proving the value does not exist in the list. Whenever bot > top, the process stops and the search is deemed unsuccessful.

7. What is the maximum number of probes necessary to find the value or to determine if the value is not in the list?

   - Given N values in the list between top and bottom, after the first probe, the number of values reduce in half to N/2 values. After the second probe, the number of values (N/2) is again reduced in half -- (N/2)/2 = N/4. After the third probe, the number of values (N/4) is halved once more (N/4)/2 = N/8. After the fourth probe, the number of values N/8 is halved to (N/8)/2 = N/16 (N/16 is equal to $N/2^4$).

   - In general, after M probes, the values between top and bottom is $N/2^M$ where M represents the number of probes. The maximum number of probes necessary to either locate the value or determine if the value is not there is M times where M is the smallest number such that $2^M >= N$.

8. Develop a recursive binary search routine using a WHILE loop.

```
CONST
  maxsize = 1000; (* maximum size of array *)
TYPE
  arraytype = ARRAY [1..maxsize] OF integer;
FUNCTION Binsearch  (VAR list : arraytype; key, bot, top : integer) :
                    integer;   (* Returns zero if item is not found *)
VAR
  mid : integer;
BEGIN
  mid := (bot + top) DIV 2;
  IF bot > top THEN
    Binsearch := 0
  ELSE
  IF key = list[mid] THEN
   Binsearch := mid
  ELSE
   IF key < list[mid] THEN
    Binsearch := Binsearch (list, key, bot, mid - 1)
   ELSE
    Binsearch := Binsearch (list, key, mid + 1, top)
END;
```

<u>Summary:</u>

1. Why is a binary search a more efficient and less time-consuming searching technique than the sequential search when searching through an ordered list?

   . Where the sequential search requires each element in the list to be examined until the desired element is found, the binary search uses an algorithm which discards half the list once it determines the desired element resides in the other half. Thus, a sequential search requires a maximum of ten probes for a 10-element list, and the binary search only requires four.

2. How do you determine the maximum number of probes required in a binary search? (Simply raise 2 to the smallest power which will give a result greater than or equal to the number of elements in the list; that smallest power is the maximum number of probes.)

<u>Exercises:</u>

1. Using a binary search, determine the maximum number of probes necessary to find a desired value (key) or no value in a list of:

   a) 25 elements ($2^5 > 25$ = 5 probes)
   b) 100 elements ($2^7 > 100$ = 7 probes)
   c) 1000 elements ($2^{10} > 1000$ = 10 probes)

2. The following questions are from the sample A.P. exam:

   . Suppose that L is a list of distinct numbers arranged in increasing order and that a binary search is to be used to determine whether some number X occurs in this list.

     - If L contains 127 entries and X actually occurs as the 80th entry of L, how many entries from L will be examined by the search?

       (a) 1
       (b) 3
       (c) 12.7
       (d) 80
       (e) 127

   Using the binary search with 127 elements:

Probe 1 (1 + 127) div 2 = 64th element
64th < 80th element

Probe 2 (65 + 127) div 2 = 96th element
96th > 80th element

Probe 3 (65 + 95) div 2 = 80th element
The answer is (B) since it takes 3 searches.

. If L contains 26 entries, what is the average number of entries in L that must be examined by the search?

(a) 26
(b) 13
(c) 6
(d) 5
(e) 4

. Since there are 26 entries, the maximum number of entries to be searched is five. This is because 2 to the 5th power is 32, and 32 is greater than 26. In a binary search, the average number of entries that must be examined is approximately one less than the maximum number. Therefore, the average number should be approximately 4, which is choice (e).

3. Write a non-recursive binary search function.

```
FUNCTION Binsearch  (VAR list : arraytype; key, bot, top : integer):
                     integer;
VAR
  mid : integer;
  found : boolean;
BEGIN
  found := false;
  WHILE (bot<= top) AND (NOT found) DO
    BEGIN
      mid := (low + high) DIV 2
      IF key = list[mid] THEN
        found := true
      ELSE
        IF key < list[mid] then
          top := mid - 1
        ELSE
          bot := mid + 1
    END;
  IF found THEN
    binsearch := mid
  ELSE
    binsearch := 0
END;
```

TOPIC 44

IMPLEMENTING MULTI-DIMENSIONAL ARRAYS

Performance Objectives - Students will be able to:

- declare multi-dimensional arrays
- show input and output operations on two-dimensional arrays
- explain how calculations can be performed on rows and columns of two-dimensional arrays

Vocabulary: multi-dimensional array, matrix, row, column

Motivation:

- Write a program that will, on input, assign a value of 'E'mpty or 'P'iece to each element of the first row of a checkerboard.

```
PROGRAM Checker (Input, Output);
VAR
  board : ARRAY [1..8] OF char;
  X : integer;
BEGIN
  writeln;
  FOR X := 1 TO 8 DO
    readln (board[X])
END.
```

- How can we extend this concept and assign an 'E' or 'P' to all the rows of a checkerboard, which is essentially a table of two dimensions?

Aim: How are two-dimensional and multi-dimensional arrays implemented?

Development:

1. In order to implement a two-dimensional array, it must be declared. The declaration can be included in either the VAR statement or both the VAR and TYPE statements.

   - In general, the VAR statement reads:

     ```
     VAR
     identifier : ARRAY [row range, column range] OF element type;
     ```

     - In the checkerboard example, the VAR declaration could read:

       ```
       VAR
         checkerboard : ARRAY [1..8, 1..8] OF char;
       ```

   - When declaring arrays with a TYPE statement, the format is as follows:

```
TYPE
  type_identifier = ARRAY [row range, column range] OF element type;
VAR
  identifier : type_identifier;
```

- The checkerboard might be declared as:

```
TYPE
  board = ARRAY [1..8, 1..8] OF char;
VAR
  checkerboard : board;
```

. The type identifier identifies or describes the array structure.

. The range determines the number of cells in the array. The subscript type may be either of the standard types -- boolean or character -- or any user-defined scaler type, or a subrange. The range is usually expressed in the form 1..5, min..max, or 'A'..'Z'.

- Note: Neither real nor integer can be used as a subscript type. However, any reasonably-sized subrange of the integers can be used.

. The element type describes the type of each element to be stored.

2. Declare a two-dimensional array to store the following gross receipts information:

	Franks	Burgers
Mon.	16.50	8.50
Wed.	20.00	9.00
Fri.	24.00	11.00

```
VAR
  table : ARRAY [1..3, 1..2] OF real;
```

. The first index value [1..3] refers to the rows (Mon, Wed, Fri), and the second index value [1..2] refers to the columns (Franks, Burgers).

. To access a specific element in the array, we use a subscripted variable. Table[1,2] refers to row 1, column 2 which would be the gross receipts for Monday's burgers (8.50). Table[2,1] refers to row 2, column 1 -- Wednesday's franks (20.00).

. Since array elements are accessed by location, two subscripts are needed to access elements of a two-dimensional array (sometimes called a table or matrix).

3. Write a statement that would assign to the variable X the total receipts for franks sold on Monday, Wednesday, and Friday.

```
X := table[1,1] + table[2,1] + table[3,1];
```

This statement stores 60.50 in variable X.

4. Declare a two-dimensional array using a type declaration statement. Use the same chart as before.

	Franks	Burgers
Mon.	16.50	8.50
Wed.	20.00	9.00
Fri.	24.00	11.00

. Three ways of declaring this two-dimensional array are shown below.

a)
```
TYPE
  food = ARRAY [1..3, 1..2] OF real;
VAR
  foodcost : food;
```

b)
```
TYPE
  food = ARRAY [1..3] OF ARRAY [1..2] OF real;
VAR
  foodcost : food;
```

c)
```
TYPE
  days = (mon, wed, fri);
  menu = (franks, burgers);
  food = ARRAY [days, menu] OF real;
VAR
  foodcost : food;
```

5. How can we input and output information in a two-dimensional array?

. Consider the example of students taking tests. Using the following chart, how can you input the three test grades for each of the four students?

TESTS

STUDENT	1	2	3
1	80	90	85
2	70	95	90
3	50	80	95
4	60	75	90

- Assuming that the grades were declared using a type statement

```
TYPE
  classtype : ARRAY [1..4, 1..3] OF integer;
VAR
  class : classtype;
  I, J : integer;
```

- The grades can be entered with the following nested loop.

```
FOR I := 1 TO 4 DO
  FOR J := 1 TO 3 DO
    readln (class[I,J]);
```

6. Let's trace the input of data to see how the data is read into a two-dimensional array.

I	J	class[I,J]
1	1	class[1,1]
	2	class[1,2]
	3	class[1,3]
2	1	class[2,1]
	2	class[2,2]
	3	class[2,3]
3	1	class[3,1]
	2	class[3,2]
	3	class[3,3]
4	1	class[4,1]
	2	class[4,2]
	3	class[4,3]

- From the trace, you will notice that the inner loop is being executed faster, which results in the information being read across the rows.

- How can you change the program so that information is read down the columns?

```
FOR J := 1 TO 3 DO
  FOR I := 1 TO 4 DO
    readln (class[I,J]);
```

- Once again, a careful trace will show the information is being read in as follows:

I	J	class[I,J]
1	1	class[1,1]
2		class[2,1]
3		class[3,1]
4		class[4,1]

1	2	class[1,2]
2		class[2,2]
3		class[3,2]
4		class[4,2]
1	3	class[1,3]
2		class[2,3]
3		class[3,3]
4		class[4,3]

. We can output data in a similar manner:

```
FOR I := 1 TO 4 DO
  FOR J := 1 TO 3 DO
    writeln (class[I,J]);
```

7. Using a two-dimensional array, how can the checkerboard program be completed to place an 'E'mpty or a 'P'iece on the board?

```
VAR
  checkerboard : ARRAY [1..8,1..8] OF char;
BEGIN
  FOR I := 1 TO 8 DO
    FOR J := 1 TO 8 DO
      readln (checkerboard[I,J]);
  FOR I := 1 TO 8 DO
    FOR J := 1 TO 8 DO
      writeln (checkerboard[I,J])
END.
```

8. The concept of an array can be extended beyond that of a two-dimensional array. An array with two or more dimensions is classified as a multi-dimensional array.

   . For instance, we could declare tempday as a variable that would store the temperatures of the days of a month.

```
VAR
  tempday : ARRAY [1..31] OF integer;
```

   . This idea could now be extended to an array that would store the temperatures of an entire year (all 12 months).

```
VAR
  annualtemp : ARRAY [1..12, 1..31] OF integer;
```

   - How can 30° be assigned to February 28?

```
annualtemp[2,28] := 30;
```

   - Write a statement that will print "BRR!" if the temperature on March 5th is less than 10°.

```
IF annualtemp[3,5] < 10 THEN
  writeln ('BRR!');
```

- Initialize a table by assigning 0° to every day of the year

```
VAR
  month, day : integer;
BEGIN
  FOR month := 1 TO 12 DO
    FOR day := 1 TO 31 DO
      annualtemp[month, day] := 0
END;
```

- Note: We did not account for February having only 28 days, and September, April, June, and November having 30 days.

- So far, we have only encountered a two-dimensional array.

- How can you declare an array that will store the daily temperatures from the year 1900 to 2000?

```
VAR
  centemp : ARRAY [1900..2000, 1..12, 1..31] OF integer;
```

- Since there is more than one subscript, a multi-dimensional array was created. You should also note that, although creating such an array is possible, it may not be practical on a microcomputer since centemp takes up 1001 x 12 x 31 words of memory.

- Assign a temperature of 47° to February 14, 1984.

```
centemp[1984, 2, 14] := 47;
```

Summary:

1. What is a multi-dimensional array and how is it declared?

2. How do you input data into a multi-dimensional array?

Exercises:

1. Using the test grade chart, write a program that will find the average of each student's grades, and the average class score on each of the three tests. Both sets of results should be printed.

TOPIC 45

PACKING ARRAY ELEMENTS TO REDUCE STORAGE

Performance Objectives - Students will be able to:

- explain how the usual implementation of arrays wastes storage space
- describe the techniques for packing arrays
- discuss the trade off between packing and not packing arrays
- state the restriction on using packed arrays

Vocabulary: packed array, word, pack, unpack

Motivation:

- How are an array's components stored in memory?
  - The components of an array are stored in consecutive words of the computer's memory.
  - On large computers the size of each word may be 32 bits. This means that a character (which requires 7 or 8 bits) or a boolean (which requires 1 bit) may occupy up to 32 bits. This is done because the computer can operate faster with words than it can with bits or bytes.
  - This type of array implementation may operate faster, but tends to waste a great deal of storage space.
- How can we reduce the storage space?

Aim: How can we pack array elements to reduce storage?

Development:

1. A special type of array called a packed array is available in Pascal.
   - When an array is packed, the compiler stores more than one value into each memory location, thus reducing the amount of storage space.
   - Although less storage space is used when an array is packed, it usually takes longer to access an element since the computer not only has to access the memory location containing the desired value, but it also has to isolate that value from the others within the same memory cell.
2. An array is packed when the reserved word PACKED precedes the reserved word ARRAY in its declaration. For example:

```
VAR
  ch : PACKED ARRAY[1..1000] OF char;
```

packs the elements of the array ch into a smaller memory location. Normally ch would contain 1000 words. By packing this array, the compiler might pack four elements into each word, thus reducing the array to occupy only 250 words.

- As mentioned before, although a packed array takes up less memory, it is often more time-consuming to access a particular element of a packed array. Instead of accessing each element of the array one at a time, it would be more economical to unpack the array all at once.

- Pascal provides two procedures, pack and unpack, which accomplish these ends in a single operation.

3. Suppose you had the declarations

```
VAR
  A : ARRAY [1..10] OF char;
  B : PACKED ARRAY [1..7] OF char;
```

the statement:

```
pack (A,K,B);
```

copies the elements beginning at subscript position K of array A into packed array B. The copying begins at the first subscript position of B. For example,

```
pack (A,3,B);
```

takes the elements of array A, starting at A[3], and packs each into packed array B.

A[3] → B[1]
A[4] → B[2]
A[5] → B[3]
A[6] → B[4]
A[7] → B[5]
A[8] → B[6]
A[9] → B[7]

- As the above chart shows, A[3] through A[9] were packed into array B. This packing procedure is equivalent to the following loop.

```
FOR K := 1 TO 7 DO
  B[K] := A[K+2];
```

- In general, if we had

```
VAR
  A : ARRAY [M..N] OF type_T;
  B : PACKED ARRAY [I..J] OF type_T;
```

where N - M >= J - I, then pack (A,K,B) is equivalent to:

```
FOR R := I TO J DO
  B[R] := A[R - I + K];
```

- This will pack the components A[K] through A[J - I + K] into the components B[I] through B[J]. Using the original example:

```
           (* M   N *)
VAR           ↓   ↓
  A : ARRAY [1..10] OF char;
  B : PACKED ARRAY [1..7] OF char;
                    ↑  ↑
                 (* I  J *)
```

if we pack (A,K,B) and substitute 3 for K, the loop becomes

```
FOR R := I TO J DO
  B[R] := A[R - I + K];

FOR R := 1 TO 7 DO
  B[R] := A[R - 1 + 3];
```

and it is seen that

```
B[1] := A[1 - 1 + 3] or B[1] := A[3]
B[2] := A[2 - 1 + 3] or B[2] := A[4]
...
B[7] := A[7 - 1 + 3] or B[7] := A[9]
```

- Pictorally Pack (A,K,B)

```
A[K]  A[K+1]  A[K+2]  ...  A[J-I+K]  ...  A[N]
  ↘      ↘       ↘            ↘
  B[I]   B[I+1]   B[I+2]  ...  B[J]
```

4. How do you unpack an array?

- unpack (B,A,K) copies the elements beginning at the first subscript position of packed array B into unpacked array A beginning at subscript position K.

- The process is the exact reverse of what occurs when we use pack. Using the same example as previously,

  unpack (B,A,3)

  causes the value of B to be unpacked and its components to be stored in A[3] through A[9].

```
B[1] → A[3]
B[2] → A[4]
B[3] → A[5]
B[4] → A[6]
B[5] → A[7]
B[6] → A[8]
B[7] → A[9]
```

- The following FOR/DO loop accomplishes the same result.

```
FOR K = 1 TO 7 DO
  A[K + 2] := B[K];
```

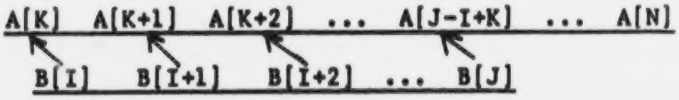
- In general, unpack (B,A,K) is equivalent to

```
FOR R := I to J DO
  A[R - I + K] := B[R];
```

and it can be seen with the following substitution

```
A[1 - 1 + 3] := B[1] or A[3] := B[1]
A[2 - 1 + 3] := B[2] or A[4] := B[2]
A[3 - 1 + 3] := B[3] or A[5] := B[3]
...
A[7 - 1 + 3] := B[7] or A[9] := B[7]
```

- Pictorally unpack (B,A,K)



5. What are some restrictions when using packed arrays?

- Packed and unpacked arrays can be used in the same way. As mentioned previously, only efficiency is affected. The only restriction is that an element of a packed array cannot be passed as a variable parameter to a function or procedure.

- Note:

a) read (A[I]) may be invalid if array A is a packed array of characters, since it involves passing an element of a packed array as a variable parameter. If this is true for a particular Pascal implementation, the following statements may be used instead

```
read (ch);
A[I] := ch;
```

b) In addition, the standard procedures pack and unpack are not provided in UCSD Pascal. These procedures are automatically done when operating on a packed variable.

Summary:

1. How does the usual implementation of an array lead to the waste of storage space?
2. What are the techniques used for packing arrays?
3. What is the trade off between packed and non-packed arrays?
4. What are some of the restrictions on using packed arrays?

TOPIC 46

IMPLEMENTING STRINGS WITH PACKED ARRAYS

Performance Objectives - Students will be able to:

- explain how strings are implemented using packed arrays
- discuss the advantages of defining a string as a packed array of char
- show how to simulate the printing of user-defined scalars

Vocabulary: string

Motivation:

- Until now, we have been using string constants such as "hello", but we have not used string data types.

- In Pascal, the word string is commonly used as a type identifier to describe a packed array of character elements.

Aim: How are strings implemented using packed arrays?

1. Character strings are stored as packed arrays of characters. For example, we could declare:

```
VAR
  name : PACKED ARRAY [1..20] OF char;
```

  - This declaration allows string constants (characters) to be assigned to the variable "name".

  - The variable name stores exactly 20 characters, including blanks. If a value assigned to name has less than 20 characters, it should be padded to 20 characters by adding blank spaces.

    - For example, if "John Doe" were assigned to name, it must be padded with blanks to make the length equal to 20 characters; that is,

```
name := 'John Doe            '; (* 12 blanks after "e" *)
```

2. When a written text is assigned to a variable name, it provides an alternative to the message that may be output by the write statement.

  - For example, we might need a program that determines if a real number is positive, negative or zero. Before we learned about packed arrays of characters, we might have assigned '+', '-', '0' to a variable called sign of type char.

  - With packed arrays of characters, we can declare:

```
sign : PACKED ARRAY [1..8] OF char;
```

so that we could now have

```
IF num > 0 THEN
  sign := 'positive'
ELSE
  IF num = 0 THEN
    sign := 'zero    ' (* 4 blanks after zero *)
  ELSE
    sign: = 'negative';
writeln  (num: 1:3, ' is ', sign)
```

- In order for the computer to distinguish between the character string 'positive' and the variable positive, the character string must be enclosed in single quotes.

- When a character string literal appears in a write statement, execution causes the computer to print the literal itself.

  - For example:

    ```
    writeln ('7 + 3 =', 7 + 3)
    ```

    causes the computer to print: 7 + 3 = 10

- The computer treats character string literals as sequences of individual characters.

3. What are the advantages of defining a string as a packed array?

   - Relational operators can be applied to packed arrays of characters of the same type. This is advantageous when we compare strings for alphabetic order. For example:

     ```
     'Mary ' < 'Molly'
     'Bonnie' < 'Bud
     'John' > 'Jack'
     'Bud' > 'Bob'
     ```

   - All of the above boolean expressions are valid and have the value of true.

4. The result of the comparisons using these operators (<, <=, >, >=, =, <>) depends on the collating sequence used in the computer.

   - There might be unexpected consequences when using upper and lower case letters. For example in ASCII, Ask < Bask < ask < bask

   - In addition, a special role should be taken for the blank. The result of a blank is dependent upon its position in the collating sequence.

     - In most ASCII character sets, the blank comes before any of the numerals and letters. Thus the following is true:

```
'Bud  '< 'Buddy'
'Stan   '< 'Stanley'
```

5. Another advantage of packed string arrays is that a text value can be given to an entire string with an assignment statement. Also, an entire string can be output at once instead of one character at a time.

   - For example if X is declared as a packed array of size 10, we can make the following assignment:

     ```
     X := 'September ';  (* one blank after 'r' *)
     ```

   - The string can then be output with a single write statement,   write (X)

   - In standard Pascal, the characters of a character string must be read one at a time. Usually a FOR loop is used to read the string:

     ```
     FOR I := 1 to 10 DO
       read (X[I]);
     ```

   - Note: In some implementations, such as UCSD, a character buffer must be used to read characters into a packed array. For example,

   ```
   FOR I := 1 TO 10 DO
     BEGIN
       read (ch);
       X[I] := ch
     END;
   ```

6. An array of character strings may be declared in any of the following forms:

   ```
   VAR
     stringlist : ARRAY [1..17] OF PACKED ARRAY [1..8] OF char;
       (* Produces a string list of 17 eight-character-long strings. *)

   VAR
     stringlist : PACKED ARRAY [1..17, 1..8] OF char;

   TYPE
     string8 = PACKED ARRAY [1..8] OF char;
   VAR
     stringlist : ARRAY [1..17] OF string8;
   ```

7. There is one major limitation in using user-defined scalar types: We are not able to read/readln or write/writeln user-defined scalar types. Given the example which follows,

```
TYPE
  fruit = (banana, apple, orange);
VAR
  dessert : fruit;
```

we cannot use the statement: `writeln (dessert);`

8. We can, however, get around this limitation in several ways.

   - By using a case statement to output the appropriate string:

```
CASE dessert OF
    banana : writeln ('banana');
    apple :  writeln ('apple');
    orange : writeln ('orange')
END;
```

   - By declaring

```
TYPE
  string6 = PACKED ARRAY [1..6] OF char;
VAR
  fr : ARRAY[1..3] OF string6;
```

   - Assigning

```
fr[1] := 'banana';
fr[2] := 'apple ';
fr[3] := 'orange';
```

   - And then using the ORD function:

```
N := ORD (dessert) + 1;
writeln (fr[N]);
```

   - A more sophisticated method is to create a string table. In addition to the user-defined scalar type, we also define "string" to be a packed array of the maximum numer of characters to be used. Then we can define "dessert" to be an array of these strings indexed by the fruit names.

```
TYPE
  fruit = (banana, apple, orange);
  string = PACKED ARRAY [1..6] OF char;
  healthfoods = ARRAY [banana..orange] OF string;
VAR
  food    : fruit;
  dessert : healthfoods;
```

     - In the main program you would initialize dessert as follows:

```
dessert[banana] = 'banana';
dessert[apple] = 'apple ';
dessert[orange] = 'orange';
```

- We could then assign one of our fruits to food and output it as follows:

```
food := apple;
writeln ('The food for today is ', dessert[food]);
```

Summary:

1. How are strings implemented using packed arrays?

2. What are the advantages of defining a string as a packed array of char?

3. How can the printing of user-defined scalar types be simulated?

TOPIC 47

IMPLEMENTING STRINGS WITH UCSD PASCAL

Performance Objectives - Students will be able to:

. discuss differences between the UCSD and standard implementation of strings
. Indicate how UCSD string functions and procedures work

Vocabulary: length, copy, pos, concat, delete, insert, UCSD string

Motivation:

. Given the global declaration:

```
TYPE
  string = PACKED ARRAY [1..80] OF char;
```

Suppose we consider a string to be terminated by a blank character, how can we write a function to determine the current length of string S?

```
FUNCTION Length (S : string) : integer;
VAR
  I     : integer;
  found : boolean;
BEGIN
  I := 80;
  found := false;
  WHILE (I > 0) AND (NOT found) DO
    IF S[I]<>' ' THEN
      found: = true
    ELSE
        I := I - 1;
  length := I
END; (* length *)
```

. Can we simplify this function in UCSD Pascal?

Aim: How are strings implemented in UCSD Pascal?

Development:

1. In UCSD Pascal, the data type, string, may be used for storing strings of characters instead of using a PACKED ARRAY OF char.

   . String variables default to 80 characters in length, but the limit can be changed by declaring type string[N] where N is an integer.

     - Explicit declarations may be made if it is known beforehand that the string will contain more or less than 80 characters.

     - The maximum length that may be declared is 255 characters.

2. How do you allocate storage for string variables?

```
VAR
  first,
  last    : string[20];
  address : string[100];
  remark  : string;
```

  . In the VAR declaration above, the variables first and last may contain up to 20 characters, address up to 100 characters, and remark up to 80 characters.

3. An assignment statement can store fewer characters than the amount declared in the declaration block. But, if more than the declared number of characters are assigned to the string, then a string overflow error occurs. Therefore, the following declarations are valid.

```
first := 'Mary';
last  := 'Bow';
```

4. String data may be read into string variables with the readln command. For example, when executing readln (first); in an interactive system, all characters typed into the keyboard before pressing the carriage return would be stored in the variable first.

  . If the user enters more than 20 characters, only the first 20 are accepted (saved). The rest are ignored. Unlike assignment statements, there is no execution error if extra characters are entered.

  . In addition, if you have the statements

```
read (first);
read (second);
```

  to read a pair of strings typed on two data lines, the first string would be stored in first, and a string of zero (no) characters (a null string) would be stored in second.

  - The reason for this is that the read statement only processes characters up to but not including the carriage return (CR).

  - The (CR) character at the end of the first data line would also terminate the second read statement without any characters being read.

  - The data lines are entered correctly if a readln(first) is used instead of read(first). When reading strings, you should always use readln, and never use read.

5. In UCSD Pascal, special functions are available for string manipulation.

  . Length(string) -- returns the current length of a string. Length returns a value of zero for the null string (an empty string).

- For example, the value assigned to variable I by the following is 14:

```
S := 'Happy Birthday';
I := length (S);
```

- Copy(source string, index, size) -- is similar to mid$ in BASIC. It returns a substring of the source string, starting at position index and consisting of size characters. Index and size must be integers.

  - For example, the following leaves 'ppy B' in variable T.

```
S := 'Happy Birthday';
T := copy (S, 3, 5);
```

- concat(string1, string2, string3,...,stringN) -- returns the string formed by concatenating all the strings in the parameter.

  - For example, 'Happy Birthday' is assigned to variable T by the statement

```
T := concat ('Happy ', 'Birthday');
```

  - Notice that the string list is composed of several source strings separated by commas. Also note that if a space were not at the end of the first string, the returned string would be 'HappyBirthday'.

- pos(pattern string, source string) -- searches a source string for a substring pattern.

  - The source string is examined from left to right to determine the first occurrence of the pattern string. If the pattern is found, the value returned is the position in the source of the first character of the pattern. If it is not found, the value returned is zero. For example

```
T := pos('at', 'What is it?'); assigns 3 to variable T.
T := pos('t', 'What is it?'); assigns 4 to variable T.
T := pos('la', 'What is it?'); assigns 0 to variable T.
```

6. UCSD Pascal also provides string procedures.

- delete(source string, index, size) -- removes size characters from source string starting with the character at position index (index and size are of type integer). For example

```
S := 'Happy Birthday';
delete(S, pos('i', S), 4);
```

  deletes the characters 'irth' from the string.

- insert(source string, destination string, index) -- inserts the source string into the destination string beginning at the index position of the destination string. For example,

```
S := 'Happy Birthday';
insert ('Sixteenth ', S, 7);
```

creates the following string: 'Happy Sixteenth Birthday'.

7. Characters in strings may be referenced by using subscripts. For example, if S has a value of 'Happy Birthday', then S[2] has a value of 'a' and S[7] has a value of 'B'.

   . If an attempt is made to reference part of a string that is not available, there will be the run-time error message: Invalid Index. Also note that the value returned by a subscripted string variable (for example S[2]) is of type char and not of type string.

Summary:

1. What are the major differences between UCSD and standard Pascal when implementing strings?

UCSD STRING PROCESSING APPLICATIONS

Performance Objectives - Students will be able to:

- describe how (UCSD) string functions can be used to decipher codes
- write meaningful programs using (UCSD) string functions
- discuss several applications of string processing

Vocabulary: text editor

Motivation:

- We are given that 'value1' and 'value2' are integers and that 'search' is a UCSD string. Furthermore, 'ch' is a variable of type char and 'onestg' is a UCSD string of length 1.
- We would like to determine if the characters contained in CH and onestg are contained in the string search. Write the Pascal statements that will set value1 and value2 equal to the character positions of the first occurrences of ch and onestg within search, respectively. A value of zero should be returned if the characters are not located in search.
  - For example, if ch = 'A', onestg = 'B', and search = 'CABBAGE PATCH' then value1 and value2 should take on the values 2 and 3, respectively.

```
value2 := pos(onestg, search);
value1 := 0;
FOR I := length(search) DOWNTO 1 DO
  IF search[I] = ch THEN
    value1 := I;
```

Aim: How to use UCSD strings.

Development:

1. The motivational problem pointed out the fact that there is a difference between a char and a UCSD string of length 1. Most of the errors that arise from using UCSD strings come from this form of type mismatch.
   - We can use the pos function with the string onestg because the pos function searches for one string within another string. We could not use it with ch since ch was not a string. Therefore, some form of character-by-character search had to be performed to look for ch. The solution given above solves the problem, although it is not very efficient.
2. In this lesson, we will present several problems that can be solved using UCSD strings and the string functions from the last lesson. The problems will be stated and a solution for each will be given. Of course, other

solutions are possible. The solutions provided will suggest a possible direction for your program development. The program solution to these problems can be developed in class or given as exercises.

3. It should be pointed out that a text editor is really a program that works with a large string. A text editor is a word processing program that allows us to save, retrieve and modify a collection of characters. These characters are letters, digits, punctuation marks, carriage returns, etc. The program that we use to create our Pascal programs is an example of a text editor. The operations that we will perform on strings can be expanded to work with larger collections of characters producing a simple text editor.

4. We would like to write a program that will decode a message. In our code each letter of the alphabet has been replaced by another letter.

   - For example, let us say that each A is replaced by a Z, each B is replaced by a Y, each C is replaced by an X, and so on. The blanks and punctuation marks are not changed. We would like our program to read in a coded message such as 'GSRH RH Z GVHG' and output the decoded message 'THIS IS A TEST'.

   - We can use two strings called 'alpha' and 'code', where alpha contains the 26 letters of the alphabet in order and code contains the coded value~~s~~ of the alphabet in order. Read in the coded string and create a new string by making the appropriate substitution.

```
PROGRAM Decode;
CONST
  alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  code  = 'ZYXWVUTSRQPONMLKJIHGFEDCBA';
VAR
  enstg, destg, onestg, charstg : string;
  int, value: integer;
BEGIN
  writeln ('Enter the encoded words and press return');
  readln (enstg);
  destg := '';
  FOR int := 1 TO length(enstg) DO
    BEGIN
      onestg := copy (enstg, int, 1);
      value := pos (onestg, alpha);
      IF value = 0 THEN
        charstg := onestg
      ELSE
        charstg := copy (code, value, 1);
      destg := concat (destg, charstg)
    END;
  writeln;
  writeln ('The decoded words are: ');
  writeln (destg)
END. (* Decode *)
```

5. Write a program to convert a simple additive Roman Numeral into its Arabic Numeral equivalent.

   . Recall that in Roman Numerals I is 1, V is 5, X is 10, L is 50, C is 100, D is 500, and M is 1000.

   . The program should read in Roman Numerals such as 'MMCIII' and output the Arabic equivalent 2103. Your program should only perform additions for this exercise. Therefore, the Roman Numeral 'XIV' should be erroneously translated into the value of 16.

```
PROGRAM Roman;
VAR
  romstg : string;
  int, value :  integer;
BEGIN
  writeln ('Enter Roman Numeral to be converted and press return');
  readln (romstg);
  value := 0;
  FOR int := 1 TO length(romstg) DO
    CASE romstg[int] OF
      'I' : value := value + 1;
      'V' : value := value + 5;
      'X' : value := value + 10;
      'L' : value := value + 50;
      'C' : value := value + 100;
      'D' : value := value + 500;
      'M' : value := value + 1000
    END;
  writeln;
  writeln ('The Roman Numeral ',romstg,' has a value of ',value)
END.
```

6. We will now write a program to handle all Roman Numeral expressions. This will involve both addition and subtraction.

   . Recall that whenever a value is followed by a larger value this first value is subtracted. Therefore, 'MCMXLVIII' becomes 1948 because the C (100) and X (10) are subtracted since they are followed by larger values M (1000) and L (50), respectively. Your program should read in a Roman Numeral expression and output the equivalent Arabic Numeral.

```
PROGRAM Roman2;
VAR
  romstg: string;
  int, first, second, value : integer;

FUNCTION Romvalue (ch : char) : integer;  (* returns the integer value
                  for a valid Roman Numeral digit *)
BEGIN
  CASE ch OF
    'I' : romvalue := 1;
    'V' : romvalue := 5;
```

```
        'X' : romvalue := 10;
        'L' : romvalue := 50;
        'C' : romvalue := 100;
        'D' : romvalue := 500;
        'M' : romvalue := 1000
      END
    END; (* Romvalue *)

    BEGIN
      writeln ('Enter Roman Numeral to be converted and press return');
      readln (romstg);
      value := 0;
      first := romvalue(romstg[1]);
      FOR int := 2 TO length(romstg) DO
        BEGIN
          second := romvalue (romstg[int]);
          IF first>= second THEN
            value := value + first
          ELSE
            value := value - first;
          first := second
        END;
      value := value + first;
      writeln;
      writeln ('The Roman Numeral ', romstg,' has a value of ', value)
    END.
```

7. Write a program that reads in a string containing up to 20 words and divides them into strings of one word each. Use an array of 20 strings to keep the words.

```
    PROGRAM Breakup;
    VAR
      original, newstg : string;
      stg : ARRAY [1..20] OF string;
      index, value: integer;
    BEGIN
      writeln ('Enter your original multi-word string');
      readln (original);
      newstg := original;
      index := 0;
      WHILE length(newstg) > 0 DO
        BEGIN
          index := index + 1;
          value := pos (' ',newstg);
          IF value = 0 THEN
            BEGIN
              stg[index] := newstg;
              newstg := ''
            END
          ELSE
            BEGIN
```

```
                stg[index] := copy (newstg, 1, value-1);
                delete (newstg, 1, value)
              END
          END;
        writeln;
        writeln ('The original string was: ');
        writeln (original);
        writeln;
        writeln ('The new strings are: ');
        FOR value := 1 TO index DO
          writeln (stg[value])
      END. (* Breakup *)
```

Summary:

1. Is a string of length 1 the same as a char? Explain.

2. What is a text editor?

Exercises:

1. In this lesson we have written a program to decode a message. Write a program that will encode a message; that is, read in a message and output its coded equivalent.

2. Write a program that will both encode and decode messages.

3. Write a function that will determine the number of words contained in a given string.

TOPIC 49

STANDARD PASCAL STRING PROCESSING

Performance Objectives - Students will be able to:

- describe the difficulties in implementing strings in standard Pascal
- discuss methods for implementing strings in standard Pascal
- write string functions and procedures for standard Pascal

Vocabulary: dynamic structure, static structure, delimiter

Motivation:

- Describe the actions, if any, of readln (stg) and write (stg) if

  - stg is a UCSD string

    Readln (stg) will input a string value. You can enter a stream of characters followed by a carriage return. Write (stg) will print out the value of string stg using as many spaces as there are characters in stg.

  - stg is a PACKED ARRAY [1..80] OF char

    Readln (stg) is an invalid statement. Write (stg) will print out the value of stg using 80 character positions.

  - stg is an ARRAY [1..80] OF char

    Readln (stg) and write (stg) are both invalid statements.

Aim: How to implement strings in standard Pascal.

Development:

1. The motivational problem should remind us of the differences between an ARRAY OF char, a PACKED ARRAY OF char, and a UCSD string. The main difference is in the way that the lengths of these structures are viewed.

   - The two ARRAYs are considered to have a fixed, or static, length of 80 characters. The UCSD string, on the other hand, has a varying, or dynamic, length that varies depending on how many characters are in the string at any given moment.

   - A truly dynamic structure can vary to almost any size while the program is being executed. Our UCSD string is not truly dynamic since it has a size limit of 80 (or possibly up to 255), but as long as it does not exceed this limit it behaves as a dynamic structure.

   - The dynamic behavior of strings is the largest obstacle to overcome if we want to implement strings in standard Pascal. We must also write string functions and procedures if we want the flexibility that the UCSD strings have to offer. With very few limitations, we can create strings in standard Pascal that behave very much like UCSD strings.

2. One simple method of implementing strings is to use an ARRAY OF char or a PACKED ARRAY OF char for each string. We dimension the structure for a certain number of characters and pad the unused portion of the array with blanks. We could then write functions and procedures like those available in UCSD Pascal.

   - The major problem with this solution is that our strings could not have blanks as their last character(s). This implementation will not be developed since it is often necessary or desirable to have a string end in a blank or several blanks.

3. We will choose an implementation which is an extension of the above. We will use a PACKED ARRAY OF char that is dimensioned for one more character than our string limit. Note that arrays could have been used instead of packed arrays.

   - We will place a special character, called a delimiter, after the last valid character of our string. Our delimiter will be an asterisk ('*') and our string will consist of all the characters in the array up to the character preceding the first '*'. An empty string will consist of an array with an '*' in the first position.

   - Since we are ending the string before the first '*', our strings will not be able to contain asterisks. If we did not want our strings to have this limitation, we could have chosen to end our strings before the last '*' instead of the first '*'. We did not choose this more flexible option since it would have made the writing of functions and procedures more confusing to the students.

   - For strings of length 80 we can use the following declaration:

```
TYPE
  nstring = PACKED ARRAY [1..81] OF char;
```

4. In UCSD Pascal we can assign a string with the value of the empty string (stg := ''). Since this statement is illegal if stg is of type nstring, we must write a procedure to perform this assignment. From our previous description of an empty string it should be clear that the following procedure, called nempty, will perform this action.

```
PROCEDURE Nempty  (VAR stg : nstring);  (* sets stg to an empty string *)
  BEGIN
    stg[1] := '*'
  END;  (* Nempty *)
```

- Notice that we only assigned a value to the first character of the array. In all the subsequent procedures and functions that we write we must keep this in mind. Any attempt to use characters beyond the '*' produces unexpected results since we are not padding the arrays.

5. We can assign values to our strings using string constants since we used packed arrays in our implementation instead of arrays, but our string constants must be 81 characters in length. The delimiter must be present in these assignments so we can determine the length of the string.

   - We will not be able to enter string values using readln(stg) as in UCSD Pascal, but we can write a procedure which will allow us to enter a value for stg. The following procedure, called Nenter, is one such procedure.

```
PROCEDURE Nenter  (VAR stg : nstring);
VAR
  ch : char;
  int : integer;
BEGIN
  int := 1;
  writeln ('Enter string value with "*" at end: ');
  REPEAT
    read (ch);
    stg[int] := ch;
    int := int + 1
  UNTIL (ch = '*') OR (int > 81);
      (* Check for possible overflow *)
  IF int > 81 THEN
    stg[81] := '*';
  writeln
END; (* Nenter *)
```

6. We could use write (stg) for our strings since they are packed arrays, but the output would be disastrous. The output would include our string value, the delimiter and the rest of the unused (and uninitialized) characters in the array. We will, therefore, need a procedure such as Nprint, which follows, to output our strings.

```
PROCEDURE Nprint (stg : nstring);
VAR
  int : integer;
BEGIN
  int := 1;
  WHILE stg[int]<>'*' DO
    BEGIN
      write (stg[int]);
      int := int +1
    END
END; (* Nprint *)
```

7. Our next step is to write a length function, called Nlength, that returns the length of a string.

```
FUNCTION Nlength  (stg : nstring) : integer;
VAR
  int : integer;
BEGIN
  int := 1;
  WHILE stg[int]<>'*' DO
    int := int + 1;
  Nlength := int - 1
END;  (* Nlength *)
```

8. We have two problems when we try to implement the UCSD concat function.

   - First, the UCSD concat function can handle a variable number of parameters. Our procedures and functions must have a fixed number. We will have to handle the concatenation of two strings and use this process repeatedly for the concatenation of more than two strings.

   - Second, we are restricted in that our functions cannot take on the value of nstring. Recall from our work with functions that our functions can only take on the value of the four standard types, user-defined scalars and subranges. This means we cannot have a function, like the UCSD concat, which returns the value of a string (or PACKED ARRAY in our case).

   - We can get around this obstacle by writing a concatenation procedure instead. The procedure will have a variable parameter to return the newly created concatenated string. Our parameters will be the name of the new string along with the two strings which will make up the new one. The following procedure, called Nconcat, illustrates this.

```
PROCEDURE Nconcat (VAR stg : nstring; st1, st2 : nstring);
VAR
  int, len : integer;
BEGIN
  stg := st1;
  int := 1;
  len := Nlength(st1);
  WHILE (st2[int]<>'*') AND ((len + int)<= 80) DO
    BEGIN
      stg[len + int] := st2[int];
      int := int + 1
    END;
  stg[len + int] := '*'
END; (* Nconcat *)
```

9. The three remaining UCSD functions and procedures are pos, delete, and copy. The pos function has been implemented by the function called Npos. This function is included in the program which follows.

   - The implementations of delete and copy are left as exercises. By the previous remarks you will realize that the UCSD copy function will have to be rewritten as a procedure instead of a function.

   - The program that follows contains all of the functions and procedures previously developed, as well as the Npos function. It also contains, as the main program, some test statements that can be used to insure that the various procedures and functions perform properly.

```
PROGRAM Nustring;
CONST
  maxlength = 80;
  maxsize = 81;
TYPE
  nstring = PACKED ARRAY [1..maxsize] OF char;
VAR
  stg1, stg2, stg3 : nstring;
  ch : char;

PROCEDURE Nempty (VAR stg : nstring);
BEGIN
  stg[1] := '*'
END; (* Nempty *)

PROCEDURE Nenter (VAR stg : nstring);
VAR
  ch : char;
  int : integer;
BEGIN
  int := 1;
  writeln ('Enter string value with "*" at end: ');
  REPEAT
    read (ch);
    stg[int] := ch;
    int := int + 1
  UNTIL (ch = '*') OR (int > maxlength + 1);
      (* Check for possible overflow *)
  IF int > maxlength + 1 THEN
    stg[maxlength + 1] := '*';
  writeln
END; (* Nenter *)

PROCEDURE Nprint (stg : nstring);
VAR
  int : integer;
BEGIN
  int := 1;
  WHILE stg[int]<>'*' DO
    BEGIN
      write (stg[int]);
```

```
        int := int + 1
      END
    END; (* Nprint *)

  FUNCTION Nlength (stg : nstring) : integer;
  VAR
    int : integer;
  BEGIN
    int := 1;
    WHILE stg[int]<>'*' DO
      int := int + 1;
    Nlength := int - 1
  END; (* Nlength *)

  PROCEDURE Nconcat (VAR stg : nstring; st1, st2 : nstring);
  VAR
    int, len : integer;
  BEGIN
    stg := st1;
    int := 1;
    len := Nlength(st1);
    WHILE (st2[int]<> '*') AND ((len + int)<= maxlength) DO
      BEGIN
        stg[len + int] := st2[int];
        int := int + 1
      END;
    stg[len + int] := '*'
  END; (* Nconcat *)

FUNCTION Npos (sub, stg : nstring) : integer;  (* returns the character
              position of the first occurrence of string sub in the
              string stg.  A value of 0 will be returned if there is no
              occurrence.  Search for empty string sub returns a value
              of 1. *)
  VAR
    sublength, stglength, index, int : integer;
    match, found : boolean;
  BEGIN
    sublength := Nlength(sub);
    stglength := Nlength(stg);
    found := FALSE;
    index := 1;
    WHILE (NOT found) AND ((index + sublength -1)<= stglength) DO
      BEGIN
        match := TRUE;
        int := 1;
        WHILE match AND (int<= sublength) DO
          BEGIN
            IF sub[int]<>stg[index + int - 1] THEN
              match := FALSE;
            int := int + 1
          END;
        IF match THEN
        found := TRUE
      ELSE
        index := index + 1
    END;
  IF found THEN
    Npos := index
  ELSE
    Npos := 0
END; (* Npos *)

BEGIN
  writeln ('Empty string test');
  Nempty (stg1);
  writeln ('Length of string 1 is ', Nlength (stg1));
  write ('String #1: ');
  Nprint (stg1);
  writeln ('---');  (* --- after string to check for blanks *)
  writeln ('Test Nenter procedure');
  Nenter (stg1);
  writeln ('Length of string 1 is ', Nlength (stg1));
  write ('String #1 :');
  Nprint (stg1);
  writeln ('---');
  writeln ('Test Nconcat procedure');
  writeln ('Enter first string: ');
  Nenter (stg1);
  writeln ('Enter second string: ');
  Nenter (stg2);
  Nconcat (stg3, stg1, stg2);
  writeln ('First string: ');
  Nprint (stg1);
  writeln ('---');
  writeln ('Second string: ');
  Nprint (stg2);
  writeln ('---');
  writeln ('Concatenated string: ');
  Nprint (stg3);
  writeln ('---');
  write ('Their respective lengths are: ');
  writeln (Nlength (stg1), ' ', Nlength (stg2), ' ', Nlength (stg3));
  REPEAT
    writeln ('Enter target string: ');
    Nenter (stg1);
    writeln ('Enter source string: ');
    Nenter (stg2);
    writeln ('The character position is ', Npos (stg1, stg2));
    writeln;
    write ('Do you want to test Npos again (Y/N)? ');
    read (ch);
    writeln
  UNTIL ch<>'Y'
END. (* Test Program *)
```

Summary:

1. How do UCSD strings differ from ARRAYs (PACKED ARRAYs) OF char?

2. What are two methods of implementing strings in standard Pascal?

3. Why do some UCSD string functions have to be rewritten as procedures?

Exercises:

1. Implement the UCSD delete procedure calling it Ndelete. The procedure call Ndelete(stg,A,B) should delete B characters from stg starting at character position A.

2. Rewrite the UCSD copy function as a procedure called Ncopy. The procedure call Ncopy(stg1,stg2,A,B) should assign a value to stg1. The value should be a string with B characters taken from stg2 by starting at the A character position.

3. Write a boolean function, called Nequal, to test the equality of two strings. The function should return a value of TRUE when the two strings are equal and a value of FALSE, otherwise.

4. Write a boolean function, called Nless, to test whether one string is less than another. The function Nless(stg1,stg2) should be TRUE when stg1 is less than stg2 and FALSE, otherwise. Recall that 'A' < 'AB' and '81' < '9'.

5. Use the previous two functions to write a greater than or equal to function called Ngtet.

TOPIC 50

SETS IN PASCAL AND THEIR OPERATIONS

Performance Objectives - Students will be able to:

- write set declarations
- define and implement set operations: union, intersection, difference and IN
- explain how the relational operators can be used to compare sets

Vocabulary: set declaration, IN, base type, set, union, intersection, difference, empty set, scalar type, subrange type

Motivation:

- Given the declaration:

```
VAR
  word : ARRAY [1..5] OF char;
```

where word represents a 5-letter word of capital letters, write a program fragment to determine the number of vowels it contains.

- Possible solution:

```
numvowel := 0;
FOR I := 1 TO 5 DO
  IF (word[I] = 'A') OR (word[I] = 'E') OR (word[I] = 'I') OR (word[I]
  = 'O') OR (word[I] = 'U') THEN numvowel := numvowel + 1;
```

- Can we express the boolean part of the IF/THEN statement above more concisely?

Aim: How can sets be used to simplify operations on collections of data items?

Development:

1. In Pascal, a set is a structured data type in which elements must be of the same type, called the base type.
   - The base type can be any scalar or subrange type.
2. A set workdays with base type days can be declared in the two following ways:

```
TYPE
  days = (Mon, Tues, Wed, Thurs, Fri);
```

a)

```
VAR
  workdays : SET OF days;
```

b)

```
VAR
  workdays : SET OF mon..fri;
```

71

3. How many elements can be in a set?

   - While a set in mathematics can have from 0 (the null or empty set) to an infinite number of elements (infinite sets), a set in Pascal must be finite.

   - Limitations on the number of elements varies with the implementation and should be determined beforehand to avoid errors.

4. How can you initialize a set to contain only vowels? to be empty?

```
vowels := ['A', 'E', 'I', 'O', 'U'];

emptyset := [];
```

   - Set notation in Pascal uses square brackets [], instead of the curly braces used in mathematics.

5. What operations can be performed on sets?

   - Review the set operations of intersection, union and difference:

     - The intersection of two sets is a new set containing the elements common to both sets.

     - The union of two sets is a new set containing the elements that are in either set or in both sets.

     - The difference of two sets is a new set containing all the elements of the first set which are not elements of the second set.

   - Introduce their Pascal symbols:

     '*' for intersection
     '+' for union and
     '-' for difference.

   - Given the following sets:

     A = [red, white, blue]   and   B = [blue, green]

     - What is the intersection of A and B? (A * B = [blue])

     - What is the union of A and B? (A + B = [red, white, blue, green])

     - What is the difference between A and B? (A - B = [red, white])

     - What is the difference between B and A? (B - A = [green])

6. Indicate that the motivation problem could have been facilitated by the set operation of inclusion.

   - Individual elements of a Pascal set cannot be accessed, nor can write and read procedures be used with sets. However, a particular element's membership in a set can be determined by using the operator "IN".

     - For example, IF ch IN ['A','E','I','O','U'] is equivalent to IF (ch = 'A') OR (ch = 'E') OR (ch = 'I') OR (ch = 'O') OR (ch = 'U').

   - Thus a more concise solution to the motivation problem would be

```
numvowel := 0;
FOR I := 1 TO 5 DO
  IF word[I] IN ['A','E','I','O','U'] THEN
    numvowel := numvowel + 1;
```

7. How does the set of vowels compare with the set of letters in the alphabet? (It is a proper subset.)

   - How can we indicate that relationship in Pascal?

   - Introduce the relational operators which can be used to compare sets.

     - set 1 = set 2 means the sets are identical. Neither the order nor the repetition of elements have an effect on the comparison. So, the following statements are true: ['A', 'E', 'I'] = ['I', 'A', 'E'] and ['A', 'E', 'A'] = ['A', 'E'].

     - set 1 <> set 2 means at least one element in either set is not present in the other set.

     - set 1 <= set 2 means set 1 is a subset of set 2.

     - set 1 < set 2 means set 1 is a proper subset of set 2

     - set 1 >= set 2 means set 1 is a superset of set 2 (set 2 is a subset of set 1).

     - set 1 > set 2 means set 1 is a proper superset of set 2

Applications:

1. Given set A = ['X', 'Z'], Write code to add element 'Y' to set A, and delete element 'Z' from set A.

2. Write code to initialize: the variable "alphabet" to contain the letters of the alphabet; the variable "setofvowels" to contain the vowels; and the variable "setofconsonants" to contain the consonants.

   - Note: The set of consonants is the complement of the set of vowels if alphabet is the universal set.

3. Given:

```
allmonths :=  [jan, feb, mar, apr, may, jun, jul, aug, sep, oct,
               nov, dec];
summonths := [jun..aug];
winmonths := [dec, jan..mar];
jmonths := [jan, jun, jul];
```

. Indicate by roster the following sets:

- summonths + jmonths
- summonths * jmonths
- summonths - jmonths
- jmonths - summonths
- summonths * winmonths

4. Using the sets defined in the third example, find the values of the following boolean expressions:

. summonths<= allmonths
. winmonths > jmonths
. jmonths * summonths = jmonths

5. If A := ['A'..'N'] and B := ['K'..'Z'], find

. A + B
. A - B
. B - A
. A * B

Summary:

1. How are sets defined in Pascal?

2. How is a set declared?

3. What operations can be performed on sets?

4. How can sets be used to facilitate programming?

5. What are the limitations of sets?

Exercises:

1. Consider the following program statement:

```
CASE grade OF
  'A', 'B' : write ('good work');
       'C' : write ('average work');
  'D', 'F' : write ('poor work')
END (* case *)
```

- If grade does not contain an 'A', 'B', 'C', 'D' or 'F' you might get a runtime error or the statement might be skipped depending on your compiler. Add a statement that will prevent this from happening.

2. Write a program that will replace all vowels in a line of text with $, print the line and print the list of consonants in alphabetical order; for example, if input is "All men are created equal" then output will be:

```
"$ll m$n $r$ cr$$t$d $q$$l"
"the list of consonants is cdlmnqrt".
```

TOPIC 51

ILLUSTRATION OF SET PROPERTIES

Performance Objectives - Students will be able to:

- discuss the advantages and disadvantages of using sets
- explain how the elements in a set can be printed
- indicate how sets can be simulated using a boolean array

Vocabulary: union, intersection, difference

Motivation:

- Create a program fragment to write the uppercase letters of W, where W is declared to be a set of characters and ch is a character variable.

```
FOR ch = 'A' TO 'Z' DO
  IF ch IN W THEN
    write (ch);
writeln;
```

Aim: How can sets facilitate writing a program?

Development:

1. Write a program that reads a line of uppercase text and writes that line replacing all vowels by a '$'. Also, write all the consonants used in the line in alphabetic order. Use sets to facilitate the program.

```
PROGRAM Setuse (input, output);
TYPE
  universe = SET OF 'A'..'Z';
VAR
  alpha, vowels, conson, W (* set of consonants used *) : universe;
  ch : char;
BEGIN (* initialize set variables before their use *)
  alpha := ['A'..'Z'];
  vowels := ['A', 'E', 'I', 'O', 'U'];
  conson := alpha - vowels;
  W := [ ];
  WHILE NOT eoln DO
    BEGIN
      read(ch);
      IF ch IN vowels THEN
        write ('$')
      ELSE
        IF ch IN conson THEN
          BEGIN
            W := W + [ch];
            write (ch)
          END
        ELSE
```

```
          write (ch)
       END; (* while not eoln *)
     writeln; (* carriage return *)
     writeln;
     FOR ch := 'A' TO 'Z' DO
       IF ch IN W THEN
           write (ch);
     writeln
   END.  (* Setuse *)
```

- Emphasize that although the elements in a set cannot be written directly, they can be written by using a command similar to that above.

- Indicate that conson is most easily initialized as a set difference, since it is the complement of vowels where the universe is alpha.

- Emphasize that set variables should be initialized before use.

2. Have the class do the following with and without the use of sets. Write a program fragment that reads a line of characters and translates lowercase into uppercase characters.

   - Without sets

```
WHILE NOT eoln DO
  BEGIN
    read (ch);
    IF(ch > = 'a')AND(ch < = 'z') THEN
      ch := chr(ord(ch) - ord('a') + ord('A'));
    write (ch)
  END;
```

   - With sets

```
WHILE NOT eoln DO
  BEGIN
    read (ch);
    IF ch IN ['a'..'z'] THEN
      ch := chr(ord(ch) - ord('a') + ord('A'));
    write (ch)
  END;
```

   - Note that the algorithm chr(ord(ch) - ord('a') + ord('A')) does not depend on the exact value returned by the calls to ord but merely assumes that the lowercase and uppercase alphabets are contiguous and in normal alphabetic order.

   - Indicate that although both program fragments above accomplish the same end, the second fragment is easier to read.

3. Demonstrate how the following boolean implementation of sets can be used to implement the set data structure in any language. (Note: this type of simulation is also useful when the number of elements in a set exceed the number of elements that the compiler can accommodate.)

. Given a universal set T. Any subset S of T can be represented by an array A in the following way:

- put the dimension of A equal to the number of elements in set T so that each index of array A is associated with an element of set T.

- inclusion of each element in S can then be indicated by assigning a value of 1 or 0 to the appropriate position in A. (Let 1 indicate that an element belongs to S and 0 indicate that it doesn't belong.)

- For example, if T = ['A'..'E'], then array A['A'..'E'] (or A[1..5]) can be used to represent any subset S of T in the following way: If S = ['C','E'], then S can be represented as:

Array A

index	value
'A'	0
'B'	0
'C'	1
'D'	0
'E'	1

or

Array A

index	value
1	0
2	0
3	1
4	0
5	1

Note that ['C', 'E'] would be represented in the same way as ['E', 'C'].

4. In the above example, how many subsets of T are possible?

- Since each of the five indices in the array can have either of two associated values (0 or 1) there are 2^5 sets of type T. In general, if the base type has N values there are 2^N subsets which can be represented.

5. If I = ['C', 'E'] and J = ['D', 'E'] how does I + J get implemented with the above array representation?

- The union of two sets in this implementation is the array (of the same length) which contains a 1 in every position in which either I or J (or both) contain a 1. For example,

Set I + Set J = Set I + J

index	value
'A'	0
'B'	0
'C'	1
'D'	0
'E'	1

+

index	value
'A'	0
'B'	0
'C'	0
'D'	1
'E'	1

=

index	value
'A'	0
'B'	0
'C'	1
'D'	1
'E'	1

- What boolean operation does this remind you of?

- Note that union is an outgrowth of the boolean operation of disjunction (OR) in which 1 represents true and 0 represents false.

8. Similarly I * J is analogous to the boolean operation of conjunction (AND) and I * J results from the array which contains a 1 in every position in which both I and J have a 1.

Set I * Set J = Set I * J

Set I

index	value
'A'	0
'B'	0
'C'	1
'D'	0
'E'	1

*

Set J

index	value
'A'	0
'B'	0
'C'	0
'D'	1
'E'	1

=

Set I * J

index	value
'A'	0
'B'	0
'C'	0
'D'	0
'E'	1

Summary:

1. What are two advantages (readability and the ability to delete elements with the operation of set difference) and one disadvantage (lack of direct access) of using a SET structure to determine if an element belongs to a particular selection of data?

2. How can we print the elements in a given set?

3. How can sets be simulated using a boolean array?

Exercises:

1. Write a program to accept words of up to ten characters and determine if those words can be spelled using only the letters A, D, E, I, N, O, P, R, S, T.

2. Write a program that determines what percentage of letters in a text are vowels.

3. If set type T = SET OF 'A'..'E', how can you represent the following sets using a boolean array:

   a) the universal set
   b) [A, E]
   c) [] (the empty set)

4. Show how you could represent a set of integers between 1 and N by a boolean array.

5. Write routines to implement the union, intersection and difference operations for sets represented by boolean arrays.

6. Write routines to implement the tests for set equality and element membership for sets represented by boolean arrays.

TOPIC 52

PASCAL RECORDS

Performance Objectives - Students will be able to:

- state how records differ from arrays
- write record type definitions
- declare arrays of records

Vocabulary: RECORD, field

Motivation:

- A baseball team wishes to store the following data about a player on the computer for future programming:
  - The player's first name, last name, number of years playing, batting average, and if he/she is a starter.
  - How would you suggest we store this data for use in a program?

Aim: How do we store related data of different types for use in a program?

Development:

1. What are the types of the required data if we are given?

```
TYPE
  name = PACKED ARRAY [1..10] OF char;
```

   - Solution:

```
first, last : name;
yrs : integer;
ave : real;
starter : boolean;
```

2. What problems arise when attempting to store data of different types?
   - If we store each data item in a separate variable, we are not holding the data for each player (i.e. related data) together.
   - We cannot store the data in an array because we are dealing with data of different types.

3. Although we can store the data in five separate arrays, for example:

```
last, first : ARRAY [1..numplayers] OF name;
yrs : ARRAY [1..numplayers] OF integer;
ave : ARRAY [1..numplayers] OF real;
starter : ARRAY [1..numplayers] OF boolean;
```

the data for each player is not being maintained in the same area.

- It is very easy to break up the data for any one player since the only thing relating the data in one array to that in another is the (hopefully) same index. Lose the index and you lose the relationship.

- As a result, the management and updating required for any change in the team or player information becomes the programmmer's responsibility. This overhead may be too high a price to pay.

- Finally, a change for one player involves manipulating five arrays.

4. To store all of the data for one player in the same area, we need the structured data type called a record, which is a structured data type with a finite number of components (fields), not necessarily of the same type.

   - Defining a record is done in the TYPE declaration section. Simply name the record and declare the field(s) and the type of each field. For example, we may define a record called teammember as follows:

```
TYPE
  name = PACKED ARRAY [1..10] OF char;
  teammember =
    RECORD
      first,
      last : name;
      yrs : integer;
      ave : real;
      starter : boolean
    END;
```

   - Note: The final field of the record is followed by an END statement. This marks the end of the record's fields.

   - Using a record allows us to keep all of the information on one player in the same area.

5. In the VAR declaration section of our program, we can declare:

```
player : teammember;
```

   - This one declaration actually gives us five variables:

```
player.first
player.last
player.yrs
player.ave
player.starter
```

   - Note that the fields are referenced by the record name (player) followed by a '.' and the field name. (What would be the type of each?)

6. Storing information for a team should immediately indicate the need for an array. Therefore, declare team as an array of teammember.

```
VAR
  team : ARRAY [1..numplayers] OF teammember;
```

- Numplayers, of course, would be defined in the CONSTANT definition section.

7. What did we gain by defining records and declaring arrays of records?

   - All information on a player is stored in a central location, the record.
   - The entire team's information is stored in an array.
   - The information on one player can be found by referencing team[I], where I represents the index in array team of the desired player.

Applications:

1. A hardware store wishes to keep inventory on its nails based on model number. The desired information includes: quantity on hand, price per dozen, and whether or not the nail is made of brass. How would you declare the data types to store this information?

```
VAR
  nail =
    RECORD
      quantity : integer;
      price : real;
      brass : boolean
    END;
```

2. What might be the definition and declaration portions of a program that processes a school's student records? The desired information includes student name, id number, average, class, and class standing.

```
PROGRAM School;
CONST
  maxname = 10;
  studentpopulation = 1000;
TYPE
  class = (freshman, sophomore, junior, senior);
  name = PACKED ARRAY [1..maxname] OF char;
  pupil =
    RECORD
      first, last : name;
      id, rank : integer;
      grade : class;
      ave : real
    END;
  classmate = ARRAY [1..studentpopulation] OF pupil;
VAR
  student : classmate;
```

Summary:

1. What is a record, and what is its advantage?

2. What do we call the components of a record? How do we access them?

3. How can we group together related records?

Exercises:

1. An employer wants to computerize his employee records. What declarations are necessary if each record contains the following information about each employee: name, address, telephone number, social security number and salary.

2. A collector wants to print a table of all the stamps in his collection. Each line in the table is to contain the following: country of origin, year issued and condition (mint, good, worn, frazzled). Write a program to print the table that uses an array of records. Assume a maximum of 500 records.

3. Using the declaration in Program School above, write a program that prints a list of the students who are eligible for the Senior Honor Roll at Utopia High School. For admission, a student must be at least a Junior and have achieved a minimum overall average of 85%.

4. To be eligible for Driver's Education, a student must be at least 16 years old. Write a program to search a list of student records and print the names and ID numbers of all those who are currently eligible. Assume the following declarations:

   ```
   TYPE
     date =
       RECORD
         month,
         year : integer;
       END;
     student =
       RECORD
         name : PACKED ARRAY [1..10] OF char;
         IDno : integer;
         birthday : date
       END;
   VAR
     roster : ARRAY [1..100] OF student;
     currentdate : date;
   ```

5. A politician wishes to mail flyers to all registered Democrats and Liberals living in a particular postal zone. Write a program that searches an array of voter records and outputs a list of the names and addresses of such voters. Declare the record of a voter to include his name, address and party affiliation (Democrat, Liberal, Republican, Conservative, Independent).

TOPIC 53

ASSESSING INFORMATION STORED IN RECORDS

Performance Objectives - Students will be able to:

- access data stored in a record using the '.' notation and the WITH statement
- describe how records can be compared

Vocabulary: WITH

Motivation:

- Assume our baseball team data from the last lesson is stored and used in a program. Write a procedure, called Putinfo, that will print out the data on one given player.

```
PROCEDURE Putinfo (info : teammember);
BEGIN
  writeln ('Name: ',info.first, ' ', info.last);
  writeln ('Years active: ', info.yrs :3, '  Batting average: ',
          info.ave :6:4);
  IF info.starter THEN
    writeln ('This player is a starter.')
  ELSE
    writeln ('This player is not a starter.')
END;
```

- Since records store related data, there should be a method of accessing the data without having to name the record each time a field is cited.

Aim: How are fields of a record accessed?

Development:

1. We can access a record's fields by using the '.' notation.

2. The WITH statement, however, eliminates the need to repeatedly use the "." notation for each field. The proper syntax is:

```
    WITH (record name) DO
      statement
(* statement represents either a single statement or a BEGIN/END block *)
```

3. Rewrite procedure "Putinfo" using the WITH statement.

```
PROCEDURE Putinfo (info : teammember);
BEGIN
  WITH info DO
    BEGIN
      writeln ('Name: ', first, ' ', last);
```

84

```
          writeln ('Years active: ', yrs :3, ' Batting average: ', ave
                :6:4);
          IF starter THEN
            writeln ('This player is a starter.')
          ELSE
            writeln ('This player is not a starter.')
        END (* WITH *)
    END;
```

4. The WITH structure instructs the compiler to add the record name to all valid fields within the WITH block.

5. Using the TYPE declaration:

```
  TYPE
    team = ARRAY [1..numplayers] OF teammember;
```

write a procedure, called Battingaverage. The procedure should ask the user to input the number of times at bat and the number of hits for each player on the team. It should then compute and store his average in the record (average = hits/at bats).

```
    PROCEDURE Battingaverage (VAR info: team);
    VAR
      I, atbat, hits : integer;
    BEGIN
      FOR I := 1 TO numplayers DO     (* numplayers is a global constant
                                       representing the number of players
                                       on the team *)
        WITH info[I] DO
          BEGIN
            writeln ('How many times was ', first, ' ', last, ' at
            bat?');
            readln (atbat);
            writeln ('How many hits for ', first, ' ', last, '?');
            readln (hits);
            ave := hits / atbat;
            writeln (first, ' ', last, ' has a ',ave,' batting average.')
          END (* WITH *)
    END; (* Battingaverage *)
```

. There are several things you should note about the above procedure.

- hits/atbat will return a real due to the implicit conversion of integers to reals via the '/' operation.

- The parameter info is an array of record. A record or an array of records is a valid parameter as long as it has been previously defined as a type. Info must be a VAR parameter as we are changing the values of the ave fields.

- In the WITH block, first, last and ave are all fields of the record info[I] and therefore are referenced as such. Since atbat and hits are not field names of the record, they will not have the record name added even though they are referenced within the WITH block.

6. Values can be assigned to a record's fields by:

    - assigning type-compatible expressions to each field of the record
    - assigning the value of an entire compatible record to the entire record.

    - For example, if we were given the declaration:

      ```
      VAR
        mets, yankees : team;
      ```

      the following is a valid Pascal statement:

      ```
      mets[1] := yankees[4]
      ```

    - What will this statement do?

        - The statement assigns the value of each field of yankees[4] to the corresponding field of mets[1]. This statement could be used if a player was traded from one team to another.

5. Can records be compared?

    - In UCSD Pascal, you can test the equality or inequality of structured types such as records and arrays. For example,

        - mets[1] = yankees[4]

            - This boolean expression is true if each field of mets[1] has the same value as the corresponding field of yankees[4] and false otherwise. If only one field differs, the statement is false.

        - yankees[2] <> mets[4]

            - This boolean expression is true if at least one field of yankees[2] does not have the same value as the corresponding field of mets[4], and false if all of the corresponding fields contain the same value.

    - In both standard Pascal and UCSD Pascal, we can treat a record's fields as we would any other variable of that type. We can use any expression that the type allows.

    - In UCSD, we can test the equality (and inequality) of two records of the same type. These tests are not valid in standard Pascal.

<u>Applications:</u>

1. Write a routine that prints the data on only the starting team.

```
PROCEDURE Startingteam (info : team);
VAR
  I : integer;
BEGIN
  FOR I := 1 TO numplayers DO
```

```
    WITH info[I] DO
      IF starter THEN
        Putinfo (info[I])
END; (* startingteam *)
```

2. Write a UCSD routine that will print out the information on the player with the highest average and the player with the lowest average.

```
PROCEDURE Highlow (info : team);
VAR
  high, low : teammember;
  I : integer;
BEGIN
  high := info[1];
  low := info[1];
  FOR I := 2 TO numplayers DO
    WITH info[I] DO
      BEGIN
        IF ave > high.ave THEN
          high := info[I];
        IF ave < low.ave THEN
          low := info[I]
      END; (* WITH *)
  writeln ('High average: ');
  Putinfo (high);
  writeln ('Low average: ');
  Putinfo (low);

(* The following code seqment prints a player's data in case of a tie *)

    FOR I := 1 TO numplayers DO
      WITH info[I] DO
        BEGIN
          IF (ave = high.ave) AND (info[I]<>high) THEN
            BEGIN
              writeln ('High average: ');
              Putinfo (info[I])
            END;
          IF (ave = low.ave) AND (info[I]<>low) THEN
            BEGIN
              writeln ('Low average: ');
              Putinfo (info[I])
            END
        END; (* WITH *)
    END; (*highlow*)
```

3. The above procedure works in a system where a comparison of structured data types is valid. How would we revise it for other Pascal implementations?

```
    PROCEDURE Highlow (info : teammembers);
        (* Stores highest and lowest average *)
    VAR
      high, low : real;
      I : integer;
    BEGIN
      high := info[1].ave;
      low := info[1].ave;
      FOR I := 2 TO numplayers DO
        WITH info[I] DO
          BEGIN
            IF ave > high THEN
              high := ave;
            IF ave < low THEN
              low := ave
          END; (* WITH *)
      FOR I := 1 TO numplayers DO
        WITH info[I] DO
          BEGIN
            IF ave = high THEN
              BEGIN
                WRITE ('High average: ');
                Putinfo (info[I])
              END;
            IF ave = low THEN
              BEGIN
                write ('Low average: ');
                Putinfo (info[I])
              END
          END (* WITH *)
  END; (* highlow *)
```

- Note: This second method is much more economical as it contains fewer comparisons by declaring high and low as type real.

Summary:

1. What are two techniques for accessing record fields?

2. How can you assign values to each field of a record?

3. Can records be passed as parameters to procedures and functions?

Exercises:

1. Write a program that asks for the name, student ID number, gradelevel, and five grades for each of 20 students in a high school. Compute the average of each student. Print out the name and average of each student as well as the name of each student with the highest average in each of the four grade levels.

TOPIC 54

ASSESSING DATA STORED IN HIERARCHICAL RECORDS

Performance Objectives - Students will be able to:

- explain how hierarchial records can be accessed with nested WITH structures
- describe the need for record variants
- declare records that have fixed and/or variant parts

Vocabulary: hierarchy, variant record, fixed part, variant part, tag field, tag type, field list

Motivation:

- Our baseball team wants to store additional data for each player. The data will differ for pitchers and fielders.
- For the pitchers we are not that interested in the batting information, but we do want to add the following information: earned run average, strike outs, walks, number of wins, number of losses, number of saves, number of innings pitched, and number of complete games.
- For the fielders we want to add the following information: batting average, homeruns, errors, number of runs batted in, number of winning runs batted in, number of runs scored, and the player's position.
- How would we set up this record?

Aim: How can we create record hierarchies and variant records?

Development:

1. We can use the following constant and type definitions:

```
CONST
  numplayers = 28;

TYPE
  name = PACKED ARRAY [1..10] of char;
  fieldposition = (catcher, first, second, shortstop, third, right,
                   center, left);
  playertype =  (pitch, field);
  thrower =
      RECORD
        era : real;
        so,
        walks,
        wins,
        losses,
        saves,
        ip,
            games : integer
          END;
      hitter =
          RECORD
            ave : real;
            position : fieldposition;
            hr,
            error,
            rbi,
            runscored,
            wrbi : integer
          END;
      teammember =
          RECORD
            first,
            last : name;
            yrs : integer;
            starter : boolean;
            position : playertype;
            pitcher : thrower;
            fielder : hitter
          END;
      team = ARRAY [1..numplayers] of teammember;
```

. Note: Two of the fields of our record teammember are records. This is valid.

2. How would we now use these definitions in a program?

. By declaring player to be a variable of type team,

```
VAR
  player : team;
```

the following occurs:

- player becomes an array of records.
- player[1] is the first record in the array of records.
- player[1].first and player[1].last represent the first and last name of the first player in the record. Player[1].yrs represents the number of years this player has been playing. Player[1].starter will be true if this player is a starter.
- Since the field pitcher is a record, we now have player[1].pitcher.era, player[1].pitcher.so, player[1].pitcher.walks etc. for each field of the record pitcher.
- Likewise, since the field fielder is a record, we now have player[1].fielder.ave, player[1],fielder.position, etc. for each field of the record fielder.

3. We can condense this notation by using a WITH structure.

```
WITH player[I] DO
  BEGIN
    (* work with the fields first, last, ave as these are shared by
    pitchers and fielders *)
    (* test if the player is a pitcher *)
    WITH pitcher DO
      BEGIN
         (* work with the pitcher's fields *)
      END
      (* else if the player is not a pitcher, he must be a fielder *)
    WITH fielder DO
      BEGIN
         (* work with the fielder's fields *)
      END
  END (* WITH  player[I] *)
```

4. An alternative format for the above nested WITH statements follows.

```
WITH player[I], pitcher, fielder DO
  BEGIN
    (* references to first, last, yrs, starter will append player[I].
    to the field name.  References to era, so, walks, wins, losses,
    saves, ip, games will append player[I].pitcher. to the field
    name.  References to ave, position, hr, error, rbi, runscore, wrbi
    will append player[I].fielder. to the field name.  Reference to
    any other variable will be interpreted as written. *)
  END;
```

5. Nesting can create a problem if two records have a common field. For example, if first, last, yrs were fields of pitcher and fielder rather than separate fields of player[I], referencing them within the block would lead to confusion because the compiler would be unsure of which record name to add to the fields.

   - For the above reason, it is important that you make sure all ambiguities are eliminated when using nested WITHs.

6. The above structure wastes memory because space is allocated for each field even though certain fields do not apply to all players. This could prove very costly if the array is passed as a parameter within the program (as demonstrated in the last lesson).

7. To avoid the above disadvantages, Pascal offers <u>variant records</u>.

   - A variant record has a <u>fixed part</u> (a part shared by all records like our first, last, yrs) and a <u>variant part</u> (the fields unique to the different items of the record like our era, so, ave, errors, etc.)

   - When a program uses variant records, the compiler allocates space for the largest variant and declares the fields at runtime.

first	last	yrs	start	tag (pos)	era	so	walks	win	loss	save	ip	games
					ave	hr	error	rbi	runs	wrbi	fp	(unused)

- As the above diagram illustrates, each player is assigned fields for first, last, yrs, start, and tag. Then, depending upon the value of tag, the remaining fields are allocated. In our example, a pitcher's data requires more space than a fielder's. Therefore, each variant record will be assigned enough space to accommodate a pitcher. If the player is a fielder there will be some unused space that is wasted, but the amount is far less than that of our previous record structure.

8. Revise the teammember record to more economically handle the data.

```
TYPE
  name = PACKED ARRAY [1..10] OF char;
  field position =  (catcher, first, second, shortstop, third, right,
                     center, left);
  playertype = (pitcher, fielder);
  teammember =
    RECORD            (* fixed part *)
      first,
      last : name;
      yrs : integer;
      starter : boolean;
      CASE tag : playertype OF    (* variant part *)
        pitcher : (era : real; so, walks, wins, losses, saves, ip,
                   games : integer);
        fielder : (ave : real; position : fieldposition; hr, error,
                   rbi, runscored, wrbi : integer)
    END; (*ends the record definition*)
```

9. The method used to determine the appropriate variant for a given element is known as the <u>tag field</u>. When using variant records, the following should be noted:

- The CASE statement does not have its own END statement.

- When we use variant records, the fixed part must always be defined first.

- A record definition may contain only one variant part; however, any of the variant parts may have their own variants.

- Any field name in the record must be unique. You cannot have two fields with the same name in the fixed or any variant section(s).

- Each variant consists of a tag constant (e.g. pitcher) followed by a ":" and a field list (enclosed in parenthesis).

- The <u>tag field</u> type may be any ordinal type.

- Several tag constants, separated by commas, can be used in a single variant (e.g. infielder, outfielder : (field list)).

- A variant must be provided for each constant of type fieldtype. If the variant list is empty it is denoted as '()'.

- The variant used is assigned at runtime after the tag field has been given a value. Whenever the tag field is changed, the record is assigned the corresponding variant. You should never attempt to access previously stored information from variant fields once the variant has been changed. You should enter the new data into the variant fields whenever you change the variant type by changes to the tag field.

Applications:

1. The Midwood Rental Agency will rent out trucks as well as automobiles.

   - The vehicle data required for an automobile includes: license number, state, rate of rental/day, make of automobile, model, number of doors, and color.

   - The vehicle data required for a truck includes: license number, state, rate of rental/day, make, number of axles, and payload.

   - Establish a rentals record for the Midwood Rental Agency that would store the data as stated above.

   Possible Solution:

```
TYPE
     vtype = (auto, truck);
     rentals =
       RECORD
         license : PACKED ARRAY [1..10] OF char;
         state : PACKED ARRAY [1..2] OF char;
         make : PACKED ARRAY [1..5] OF char;
         rate : real;
         CASE cartype : vtype OF
           auto :  (model, color : PACKED ARRAY [1..5] OF char; numdoor
                   : integer);
           truck : (numaxles : integer; payload : real)
       END; (* RECORD definition *)
```

Summary:

1. What advantage does the use of record variants offer?

2. What are fixed fields, variant fields and tag field?

Exercises:

1. Write a procedure, startingteam, that will take in our baseball team information (distributed by the teacher) and print out the relevant information on the eight starting fielders and the four pitchers with the best (lowest) era.

2. Write a subroutine that will take information on a player just added to the team. Ensure that there is room on the roster for the player before he/she is accepted on the team.

TOPIC 55

ACCESSING DATA STORED IN VARIANT RECORDS

Performance Objectives - Students will be able to:

- explain how the fields of a variant record can be accessed
- describe how case statements are used with variant records

Motivation:

- An artist interested in geometric figures wants to keep an inventory of all her objects according to three criteria: shape, color and size. She wishes to keep a maximum of 50 objects of one of four shapes (circle, rectangle, square or triangle) at any one time. The information she wants on each object is the color (red, yellow, blue, orange or green) and whatever is needed to compute the perimeter and area of the object.

  - How would you define a data structure to accommodate the above?
  - How would you access the information in the structure?

Aim: How do we best access information stored in a variant record?

Development:

1. To do the job we must first declare the types and variables:

```
TYPE
  shade = (red, yellow, orange, blue, green);
  figure = (circle, square, rectangle, triangle);
  object =
    RECORD
      color : shade;
      CASE tag : figure OF
        rectangle : (length, width : real);
        square : (side : real);
        circle : (radius : real);
        triangle : (s1, s2, s3 : real)
    END;
  arrayofobject = ARRAY [1..50] OF object;

VAR
  inventory : arrayofobject;
  index : integer; (* to index the array *)
```

2. What role does the case statement play within the variant record? (Case statement is a means of distinguishing between the different labels or possibilities.)

3. How would you go about accessing the data in the array of records?

- Write a procedure called Getobject that would read the information in the records of any array of type arrayofobject from index 1 to ind.

```
PROCEDURE Getobject (ind : integer; VAR info : arrayofobject);
VAR
  count : integer;
BEGIN
  FOR count := 1 TO ind DO
    WITH info[count] DO
      BEGIN
        Getshade (color);
        Getfigure (tag); (* these procedures are left as exercises *)
        CASE tag OF
          rectangle : readln (length, width);
          square : readln (side);
          circle : readln (radius);
          triangle : readln (s1, s2, s3)
        END (*case*)
      END (* WITH *)
END; (*getobject*)
```

- In this example, the case is used to determine what fields apply to a particular record based on the tag field.

- The case statement is our means of determining, and thereby using, the correct variant and its fields.

4. In the running of our program we must be able to access the different fields.

- Write a procedure called printout that will print the area and perimeter/circumference of a figure.

```
PROCEDURE Printout (info : object);
CONST
  Pi = 3.14159;
VAR
  S, area, perimeter : real;
BEGIN
  WITH info DO
    BEGIN
      Getfigure (tag);
      CASE tag OF
        rectangle :
          BEGIN
            area := length * width;
            perimeter := 2 * (length + width)
          END;
        square :
          BEGIN
            area := sqr(side);
            perimeter := 4 * side
          END;
        circle :
              BEGIN
                area := Pi * sqr(radius);
                perimeter := 2 * Pi * radius
              END;
            triangle :
              BEGIN
                S := (s1 + s2 + s3)/2;
                area := sqrt(S * (S - s1) * (S - s2) * (S - s3));
                perimeter := s1 + s2 + s3
              END
          END (* case *)
        END (* WITH *)
      writeln ('area: ', area);
      writeln ('perimeter: ', perimeter);
    END; (* printout *)
```

Applications:

1. Write the necessary declarations in order to have an array of records for a business. The employees are divided into two categories, part-time and full-time.

   . The information needed for full-time employees is: name, address, id number, weekly salary, and number of absences from work.

   . The information needed for part-time employees is: name, address, id number, hourly salary, and number of hours worked this year.

   Possible solution:

```
TYPE
  emptype = (parttime, fulltime);
  name = PACKED ARRAY [1..10] OF char;
  add = PACKED ARRAY [1..30] OF char;
  worker =
    RECORD
      first, last : name;
      address : add;
      idnum : integer;
      CASE emptag: emptype OF
        parttime : (hrlywage, hrs : real);
        fulltime : (wklysal: real; absences : integer)
      END; (* record *)
```

2. Write a procedure called Employee that will read the data for an employee.

   . Possible solution:

```
PROCEDURE Employee (VAR info : worker);

BEGIN
  WITH info DO
    BEGIN
      readname(first); (*reads the packed array*)
      readname(last); (*see above*)
```

```
          readadd(address); (*see above*)
          readln(idnum);
          readtag(emptag);  (* a procedure is needed to read the
                             ennumerated type *)
          CASE emptag OF
            parttime : readln (hrlywage, hrs);
            fulltime : readln (wklysal, absences)
          END (* case *)
        END (* WITH *)
    END; (* Employee *)
```

Summary:

1. When would you use a variant record?
2. What is the purpose of the case statement when using variant records?

Exercises:

1. Write a procedure called Color that prints a table with the number of figures in each color. The table should look like the example:

Color	Circle	Square	Triangle	Rectangle

2. Using the variant record type 'worker' from Application 1, write a procedure that will give each employee a raise. The full-time employees should receive a 10.5% raise and the part-time employees should receive a raise of $1.25 per hour.

TOPIC 56

COMPUTERS IN SOCIETY

Performance Objectives - Students will be able to:

- describe the tremendous impact computers are having on our society
- explain how to use and not abuse a computer system
- cite problems related to computers and individual rights of privacy
- discuss challenges to the security of computer systems

Motivation:

1. List some of the ways that computers are used in your school; depending on the school, some typical responses may include:

   - Computer science and computer programming courses, such as Basic, Pascal, AP Computer Science and Logo.
   - Typing classes and business courses to teach basic keyboard skills and utility programs (e.g. word processors).
   - Computer aided instruction (CAI) units in many subject areas.
   - Scheduling of student program cards, class lists, etc.
   - Printing of student program cards, class lists, etc.
   - Attendance uses: reporting attendance, sending postcards and calling the home of absentee students.
   - Report card and permanent record grade entry and storage.
   - Marking tests that are submitted on optical scan forms.
   - Library uses: cataloging books and recording books that are taken out.

Aim: What are some social and ethical implications of using computers?

Development:

1. Not too long ago, computers were not used at all in the schools. From that totally manual period we have changed to a system that is becoming more and more computer oriented. Computers have not only changed the schools, but have changed just about every field and area in our society. In this lesson we will discuss some of the ways that computers have changed our way of life and the advantages and disadvantages of these changes. We will illustrate the use of computers in many areas of society, but our discussion will not include all the major impacts that the computer has presented. You may want to add more recent developments in various areas as they present themselves. You may want to include some of these topics and other topics throughout the course where they may be more relevant.

2. What are some advantages of using computers?

   - Speed - Computers can perform many tasks and calculations at incredibly fast speed. This not only includes calculations and output, but also the ability to control operations that must be responded to within a fraction of a second.

- Less Errors - Computers perform calculations and other tasks with very few errors. This of course, does not include human errors in programming.
- Reliability - As the technology and backup systems are developing, the computers are becoming very reliable.
- Long Hours - Many large computer systems operate 24 hours a day for 7 days a week.
- Better Control - Because information can be kept up-to-date in the computer and accessed easily, this allows for better control of inventories, expenses, cash flow, factory production, etc.
- More Things Possible - Because of their fast speed and large storage, computers can perform tasks that were not possible before. Certain calculations and data accumulation could not be done without computers, because they would take years to complete or would be finished after they were needed.
- Safety - Computers can control many operations, through robotics, that are too hazardous for humans.
- Cost - Though computerization may cost a lot to implement, it will usually save money in the long run. At the same time it will make more information available which can produce more savings.
- Simulation - Many complicated situations, such as company structures, disasters, and warfare, can be simulated on computers in order to select the best options.
- Availability of Information - Once information is stored in a computer, it can be made readily available for use and analysis. If the information is entered from different sources it can be correlated to present a better overall view.
- Tedious Tasks - Computers will perform the same task over and over again without complaint and thus free humans to perform other tasks.

3. What are some disadvantages of using computers?

- Loss of Jobs - The computer has eliminated or greatly reduced the need for certain jobs. On the other hand, it has also created new jobs and fields in the computer industry.
- Retraining - Since computers are used in almost every field, many people have to be retrained in their jobs in order to use the computers in their field.
- Error Correction - When a computer system is used it is harder to correct errors because the error must be corrected twice. After it is corrected by humans it must also be corrected by the computer.
- Ignoring Human Values and Individual Differences - Computers are usually programmed to handle the common entries and do not always allow for the exceptional cases. In most cases there is no way, other than making a false entry, to handle exceptional cases properly. To avoid problems with the computer, many users do not allow their customers or employees to make any exceptions to the rules, even when an exception is warranted due to the circumstances.
- Conformity to Computer Terms and Conventions - When a computer system is used we are limited in our input responses. We must convert a situation into the computer's allowable terms. In some cases this is not only difficult, but it may not present a true picture of the situation.

- Loss of Privacy - As more and more companies and government agencies keep their records on computer systems that can be accessed by other agencies, your affairs are not always kept confidential. The situation can be worsened if the information about you is incorrect.
- Dependence - Many computer users are dependent on the computer and all work stops when the computer is down. This situation will get worse as the older employees, who had once done the tasks manually, begin to leave the industry.
- Computer Crimes - The use of computers has created many different types of computer related crimes.

4. What are some of the ways computers are used around the house?

- Home Control - Thermostats, heating, air conditioning, sprinklers, fire alarms and burglar alarms.
- Home Computers - Running utility programs and communicating to other computer systems using a modem.
- Video Games and Other Computer Controlled Toys.
- Appliance Control - Televisions, stereos, video cassette recorders, ovens, microwave ovens, washers, and dryers.
- Automobile - Newer cars are controlled by a microprocessor.

5. How have computers changed the way that certain jobs are performed?

- Supermarket checkout is done by using Universal Product Code which is read by a Bar Code Scanner. This system also manages the inventory.
- Department stores use Optical Character Readers to handle checkout and inventory.
- Airline and hotel reservations are done by computer.
- Mechanical drawings and architectural blueprints are done using a CAD (Computer Aided Design) system.
- Manufacturing steps are controlled by computers using CAM (Computer Aided Manufacturing) systems. Also, robotics is coming into many of the manufacturing steps.
- Newspapers and printers use word processors that are tied into a central system which sets up the print. This eliminates the need for manual typesetting.
- Banks use magnetic ink characters on their checks, and magnetic strips on their credit cards; many allow you to bank at home by computer. The various offices of a bank are usually tied together to allow you to bank at any one of their offices.
- Trains, subways and planes are controlled and coordinated by computer.
- Emergency services, like police and fire departments, are sent by a computer which selects the nearest available unit.
- Law enforcement agencies use data bases from agencies like the Motor Vehicle Bureau and FBI for quick identification of stolen vehicles, outstanding warrants, etc.
- Hospitals use central information on items like blood and transplant availability which is kept on computers. Much of their equipment is controlled and monitored by computer.
- Telephone calls are routed by computer using the best or shortest route.
- Payrolls are computer generated by most large companies. They even report the payroll information to the IRS by computer tape.
- Most large billing for department stores, banks, credit card companies and other businesses is done by computer.

- Most large mailings use computer generated labels. Many of them use the "personalized" messages which are also computer generated.

6. What are some of the newer uses of computers?

   - Robotics - Computer controlled robots and mechanical operations
   - Graphics - TV animation, simulations and CAD systems
   - Computer Music - Entire orchestral concerts can be simulated by a computer.
   - Voice Recognition - Computers accept voice commands
   - Pattern Recognition - Reading human handwriting
   - Bio-Technology - A new field which combines biology and engineering

7. What is a data base?

   - A data base is a collection of information. The term usually refers to a collection kept on a computer.

8. What are some of the agencies which have information about people kept in a data base?

   - Motor Vehicle Bureau - Licenses, registrations, violations
   - Internal Revenue Service - Personal income tax, business tax
   - Other Federal Agencies - Passport information, military records, Social Security information, FBI National Crime Information Center
   - Credit Bureau - Credit information

9. How do computers infringe on our privacy?

   - Since there is a lot of information about people kept in computer data bases it is relatively easy for certain people to gain access to that information.

10. Do we have any right to privacy?

    - In 1974, the Freedom of Information Act was amended and the Privacy Act of 1974 was passed. These acts allow individuals to write for copies of personal records collected by federal agencies and to correct any inaccuracies in those records. Also, within limits, an individual can control disclosures of the records to other agencies. In addition, certain types of unfair information practices are subject to penalties, but this is very restricted.

11. What are some types of crimes related to computer use?

    - Theft - Computer equipment, peripherals, supplies, software
    - Fraud - Data manipulation or program modification for gain. This gain could be money or other type of gain such as changing of student grades.
    - Embezzlement - Using computers to embezzle.
    - Vandalism - Destroying data or programs. This is usually done by youngsters, prior employees or competitors.
    - Commercial Espionage - Spying on a company by accessing its data.

12. What is meant by data integrity and computer security?

- Data integrity is maintained when the data in the computer is valid. It is possible for data to become invalid by internal uses (the software or users) or by external tampering (from unauthorized users). Computer security is maintained by not allowing any unauthorized access to the data or to the computer system. In order to maintain data integrity the computer system may have internal check routines to validate the data. Whenever data is being sent to or from the system, hash and parity codes are usually used to avoid any communication errors. In order to maintain security, most systems have user ID numbers and passwords that must be given in order to access the system. Many systems also use additional user ID numbers and passwords for accessing classified information within the system. This allows different employees to access only the information that they need for their specific job.

Summary:

1. What are some of the advantages of using computers?
2. What are some of the disadvantages of using computers?
3. What is a data base?
4. How do computers contribute to your loss of privacy?

Exercises:

1. List the uses of computers in your own home.
2. List some jobs that have been eliminated or greatly reduced because of computers.
3. List some of the new jobs that have been created by computers.
4. Pick a career that you are interested in and perform the following study. First, find out how computers are being used in that field and what special computer skills are required. Then, obtain some college bulletins and see whether or not the colleges are teaching those skills for your career.
5. Find several magazine or newspaper articles that deal with computer crime or computer misuse.

   NOTE: This could also be done for some of the other topics discussed in this lesson.

TOPIC 57

INTRODUCTION TO FILES

Performance Objectives - Students will be able to:

- indicate the need for files
- tell the difference between sequential and random access files
- write a file type definition

Vocabulary: sequential file, random access file, internal file, external file

Motivation:

- A student would like to enter the names, addresses and telephone numbers of a large group of people in a computer so that the data for any given person can be quickly retrieved upon request.
- What data structure can be used to store the necessary information?
  - An array of records is unacceptable since data entered during run-time is lost as soon as program execution is completed.
  - We need a new data structure, called a file, that retains information in secondary storage as long as needed.

Aim: How are files used to store data in secondary storage?

Development:

1. What is a file?
   - A file is a sequence of components of the same type stored in nonvolatile memory. Files are either input or output. The input file is a source of data, and the output file is a destination for data.
   - The file named INPUT serves as the source of keyboard input for the variables in a read(ln) statement, and the file named OUTPUT serves as the CRT destination of output data.
     - Both of these files are predefined as type text (file of char).
     - The printer also serves as a file to which data is output.
   - There are three other types of files:
     - Data files, which are usually stored on diskette or tape, and whose components are called records (not necessarily the structured data type of that name).
     - Batch files, which often contain a series of related tasks that the computer executes sequentially, and

- Program files, which contain the source or object codes for a computer program.

2. How do files differ from arrays?

   - All files in standard Pascal are <u>sequential files</u>, terminated by an end of file marker (EOF).
     - The access method for sequential files is analogous to accessing information stored on magnetic tape.
   - Accessing the 825th record in a sequential file requires starting at the beginning of the file and reading the 824 records that precede it. In an array, however, the record can be accessed by simply writing X[825].
     - This illustrates the random access technique by which each of the components of a collection can be accessed quickly in the same amount of time.
     - Pascal has several extensions that permit random access (e.g. UCSD Pascal's SEEK procedure).
   - Arrays contain a specified amount of elements while a file's size is unlimited.
     - A file represents the external memory of the computer and thus has unlimited size (e.g. use more diskettes or tape).

3. How are files declared?

   - In a Pascal program, a file name is a variable that must be declared. The file type declaration specifies the name by which the file is referred and its component type. The syntax is:

```
file_identifier = FILE OF component_type
```

for example:

```
TYPE
  colors = (red, blue, green);
  colorfile = FILE OF colors;
VAR
  paint : colorfile;
```

   - <u>Note</u>: A file type variable can store a sequence of any component type other than another file.

4. Write a file type declaration for the motivation problem.

```
TYPE
  string20 = PACKED ARRAY [1..20] OF char;
  string15 = PACKED ARRAY [1..15] OF char;
  string5 = PACKED ARRAY [1..5] OF char;
  location =
    RECORD
      street : string20;
      city : string15;
      state, zip : string5
    END;
  person =
    RECORD
      name : string20;
      address : location;
      telenum : string15
    END;
  personfile = FILE OF person;
VAR
  people : personfile;
```

5. What would a logical representation of such a file having two components look like?

Nick Skouras 131 Livingston St. Brooklyn New York 11201 (718)121-2121	Yvette Beck 78 Park Rd. Bronx New York 11415 (212)123-4567	(EOF)

6. How do files appear to the operating system?

   - The Pascal program does not provide the operating system with either the name by which the file is known or the type of storage device on which the file is physically located.

     - For example, file F of type integer may refer to a file whose name in the diskette directory is empnum.

   - The operating system makes the actual correspondence during run-time in a manner similar to the assignment of actual parameter values to formal parameters.

   - File variables that access external files, those which are permanently stored in auxiliary memory, must be listed as parameters in the program heading. (This is not necessary for some versions of Pascal, such as UCSD.)

     - This listing should not be done for internal files (scratch files) that only exist during program execution.

7. Why do we need files?

   - Data stored in RAM is lost after program execution or when power is turned off.

     - This volatility precludes saving data in RAM that would be needed at some future time.

  - Files are essential for efficient storage of information in a programmer-created database.

- The high cost of the computer's internal memory and its limited size also make files the appropriate data structure when a programmer works with large quantities of data.

8. How does a program achieve effective communication with data files on secondary storage?

   - To counteract the comparatively long delay that occurs from the time a program issues an Input/Output (I/O) command and the time the secondary storage device responds, the operating system provides an I/O buffer large enough to hold a block of several file records. Since the buffer is in main memory, the response is fast.

     - If the buffer holds five records, up to five read or write commands can be issued without noticeable loss in speed.

     - When the buffer is full/empty, there is a time delay while the block of data is transferred to/from auxiliary storage. If this slowdown is still too long, additional buffers can often be designed.

   - The operating system also opens a file before it is accessed so that a block of records can be brought into the buffer.

   - Similarly, the operating system closes a file that is written to so that any records that remain in the I/O buffer are not lost, but are transferred to the file.

9. How do disk files and tape files differ?

   - Disk files are stored in sectors of circular tracks which have unique addresses. A movable read/write head can quickly be positioned over any files listed in the diskette directory by noting its starting address.

     - To obtain the 200th record of the fifth file listed in the directory, the computer randomly accesses the fifth file using the directory address and sequentially reads through the first 199 records. (There are Pascal extensions that permit random access of individual records.)

   - On the other hand, tape files have no directory and can only be accessed sequentially.

     - To obtain the same record on a tape, we have to reset the tape to its beginning, read the first four files completely, and then read the 199 records of the fifth file.

   - If we have to process all the data in a single file, the sequential access method is suitable.

  - However, sequential access is inappropriate for an on-line query system such as airline reservations or banking transactions where quick on-line responses to random queries are desired.

- Another problem with the sequential access method is the need to rewrite a complete file when only one item has to be modified. For these reasons, the restriction of standard Pascal to sequential files has been strongly criticized.

- Tape files are sometimes used because they are inexpensive.

  - A large corporation might dump all files onto tape each night. These tapes are kept for a period of thirty days so that a programmer can recreate a file that may have been accidentially discarded.

- The merging of sorted transaction files with master files is also an appropriate application of tape files.

Summary:

1. How do files and arrays differ?

2. Why are files an important data structure?

3. What is the difference between the random access and sequential access methods?

Exercise:

1. Write a file declaration for a file F each of whose components is an array of ten integers.

TOPIC 58

WRITING FILES

Performance Objectives - Students will be able to:

. describe how components are written to a file
. explain the purpose of the buffer variable
. show how a file of records is created

Vocabulary: rewrite, buffer variable, put

Motivation:

. Write a program that stores 30 SAT marks into a file called Scores.

. What would be an appropriate file declaration?

```
TYPE
  satmark = 200..800;
VAR
  scores : FILE OF satmark;
```

. Note that the declaration establishes the file variable, but does not create the file itself.

Aim: How can a file be created and written to?

Development:

1. A file is created with the rewrite (filename) command.

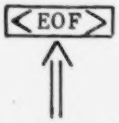
   . This command creates an empty file, called filename, with an End Of File marker (<EOF>).

   . If the filename in the rewrite command already exists, all data is lost as everything is erased before creation.

   . For example, if a program containing the statements

   ```
   VAR F : FILE OF real;
   rewrite (F);
   ```

   is executed, an empty file with an <EOF> is created:



   . The arrow pointing to the <EOF> , called the file position pointer (FPP), is a convenient symbol used to indicate the location in the file of the next file access.

2. How is a component within the file accessed?

   - When a file variable is declared, an empty buffer variable (also called file window) is automatically created.

   - The buffer variable is used to access the current component within the file. Only one component can be accessed at any given moment.

     - A buffer variable has the symbols ↑ or ∧ appended to the filename. For example, scores ↑ or scores ∧ is the buffer variable for file scores that was declared in the motivation problem. It will hold the one component of type satmark which will be accessed next.

3. How are values inserted into the empty file?

   - Values are first assigned to the buffer variable; from there, they are stored into the file with the put command.

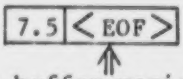
     - The put statement calls a procedure that assigns the current value in the buffer variable to the end of the file.

     - For example:

```
F↑ := 7.5;
put (F);
```

     - The first statement assigns the value 7.5 to the buffer variable F↑.

     - The second statement puts the value that is stored in the buffer variable just before the end of the file so that the file has the form:

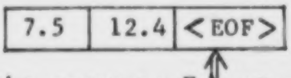


   - Since the buffer variable is undefined after a put command, it is not permissible to use two puts without an intervening assignment to the buffer variable.

     - If we now execute the statements

```
F↑ := 12.4;
put (F);
```

       the file would have the form:



   - Because the sequence F↑ := num; put(F); is so common, the procedure write (F,num); has been provided as an equivalent to the two statements.

   - All values written to a file are appended to the end of the file where the FPP is located.

5. How can we tell if we are at the proper location for writing to the file?

   - The boolean function EOF(F) can be used to check the location of the FPP. It has a value of true when the FPP has advanced beyond the last file record; otherwise, it is false.

     - If EOF(F) is not true prior to a put(F), an error will occur.

     - The statement rewrite(F) makes EOF(F) true.

6. Write the program for storing SAT scores.

```
PROGRAM Satgrades (input, output, scores);
TYPE
  satmark = 200..800;
VAR
  scores : FILE OF satmark;
  num : satmark;
  I : integer;
BEGIN
  rewrite(scores);
  FOR I := 1 TO 30 DO
    BEGIN
      write ('enter mark ',I:3,': ');
      readln (num);
      scores↑:= num;
      put (scores)
    END
END.
```

   - NOTES:

     - Since scores is a permanent external file, it is listed as a parameter in the program heading.

     - The statements readln(num) and scores↑:= num could be replaced by readln (scores↑) since scores↑ is a variable of type satmark. Or,

     - the statements scores↑:= num and put(scores) could be replaced by write(scores,num).

   - UCSD NOTES:

     - UCSD Pascal modifies the rewrite statement to include the actual name in the diskette directory associated with the file variable. For example, if the diskette file is named 'DATA:SAT1985', change the rewrite(scores) statement to:

       rewrite (SCORES,'DATA:SAT1985').

     - All files in UCSD Pascal have to be closed after file processing is completed. Close (filename) erases the file that was processed, while close (filename, lock) retains the file on diskette. Insert close (scores, lock) after the FOR statement in the above program.

Application:

1. A student would like to keep track of baseball statistics. Write a program that stores in a file records which contain a player's name, team, at bats, hits, and batting average.

```
PROGRAM Ballstats (input, output, bbfile);
TYPE
  string20 = PACKED ARRAY [1..20] OF char;
  player =
    RECORD
      name : string20;
      team : string20;
      atbat : integer;
      hits : integer;
      avg : real
    END;
VAR
  ch : char;
  bbfile : FILE OF player;
PROCEDURE Readchars (VAR st : string20);
  VAR
    I : integer;
  BEGIN
    st := '                    '; (* 20 blanks *)
    I := 1;
    WHILE (I< =20) AND (NOT EOLN) DO
      BEGIN
        read (st[I]);
        I := I + 1
      END;
    readln
  END; (* readchars *)
BEGIN (* ballstats *)
  rewrite (bbfile);  (* opens file *)
  REPEAT
    WITH bbfile↑ DO
      BEGIN
        write ('name: '); readchars (name);
        write ('team: '); readchars(team);
        write ('atbat: '); readln (atbat);
        write ('hits: '); readln (hits);
        avg := atbat/hits
      END;    (* WITH *)
    put (bbfile);
    write ('Do you have more records?  Y or N: ');
    readln (ch);
    writeln
  UNTIL (ch = 'N');
  writeln ('the baseball file has been created.')
END. (* ballstats *)
```

- UCSD NOTES:
  - Change rewrite (bbfile) to rewrite (bbfile, '< vol : diskname >').
  - Add close (bbfile, lock) after the REPEAT-UNTIL statement.
  - Program heading parameters are not needed.
  - It is simpler to use the string data type instead of packed arrays of char; the procedure would then not be used.

Summary:

1. Describe how rewrite, put, and the buffer variable are used to write to a file. (A file must be opened for writing by using the rewrite(F command. Put(F) appends the value in the buffer variable, F↑, to the end of the file.)
2. When is the buffer variable undefined?
3. What statement is equivalent to, G↑:= 150; put(G)?
4. What must be the value of the function EOF(F) before using put(F)?

Exercises:

1. Write a program that stores names, addresses and telephone numbers into a file called people.

TOPIC 59

READING FILES

Performance Objectives - Students will be able to:

- describe how components can be read from a file
- indicate how the buffer variable is used for every file access
- explain the relationship between put, get, read and write
- demonstrate how the fields of a file record can be accessed

Vocabulary: reset, get, read, (UCSD : close)

Motivation:

- Write a program that finds the average of test marks stored in a file of integers.

- If put writes to a file, what do we use to read the values already written to a file?

Aim: How can we read elements from a file?

Development:

1. How do we open a file for reading?

   - The procedure reset (filename) assigns the first component in the file to the buffer variable and sets the file position pointer (FPP) to the second component.

     - If the file is empty, the buffer variable is undefined.

2. Once the file is opened for reading, how do we access its individual components?

   - We can advance through the file with the statement get (filename).

     - The get procedure assigns that component where the FPP is located to the buffer variable and moves the FPP to the next component.

     - If the file is empty, the FPP is initially located at the EOF marker, the buffer variable cannot be assigned a value, and is therefore undefined.

     - If the file is not empty, each use of the get procedure moves us to the next position in the file.

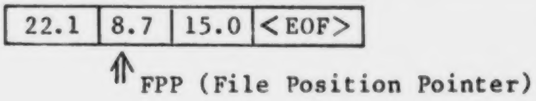
3. We can test if we have reached the end of the file with the Boolean function EOF (filename).

     - EOF is true when an empty file is reset or when we use the get procedure and the current file position (FPP) is at the end of the file; otherwise it is false.

4. Suppose we have a file F, consisting of these records of type real:

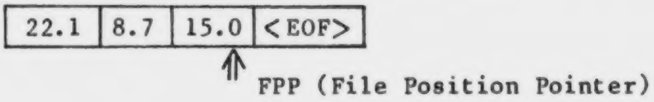
22.1	8.7	15.0	<EOF>

- When we reset (F), the buffer variable F↑takes on the value 22.1 and the FPP moves to the second record; EOF(F) is false:

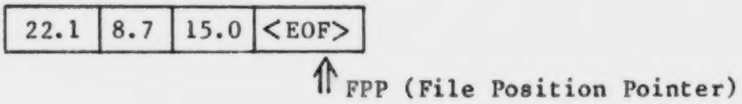


  - If we want to manipulate the value in the buffer variable, it must be assigned to a variable (e.g. num := F↑) before we move to the next file position.

- If we now execute get(F), F↑ assumes the value 8.7 and the current file position is advanced; EOF(F) is still false:



- Another application of get (F) assigns the value of 15.0 to F↑, moves the FPP and still leaves EOF(F) false:



- If we now use get(F), F↑becomes undefined and EOF(F) becomes true.

  - Since EOF(F) must be false prior to a get (F), an additional get (F) would cause an error. Therefore, one should always test EOF before calling the get procedure.

5. Because the buffer value is usually reassigned to a different variable before the next call to get, standard Pascal makes the statement

   read (F,num)   equivalent to   num := F↑; get (F)

6. In summary:

   - A file must be opened for reading with the reset (F) command.

   - Get (F) assigns the component in the current position to the buffer variable and advances the FPP to the next file position.

   - The function EOF(F) indicates when we have reached the end of the file.

   - EOF(F) must be false before the get (F) procedure is called.

7. It must be carefully pointed out that you cannot read from and write to a file at the same time.

- Just as a sequential access medium such as magnetic tape must be rewound for reading or writing, so must a Pascal sequential file be reopened for either operation.
  - A reset command is used for reading and a rewrite command for writing.
  - In a program, a particular file can be both read from and written to, but never without an intervening reset or rewrite.

Application:

1. The following program answers the motivation problem.

```
PROGRAM Average (output, scores);
VAR
  scores : FILE OF integer;
  sum, num, count : integer;
  avg : real;
BEGIN
  reset (scores);
  sum := 0;
  count := 0;
  WHILE NOT EOF(scores) DO
    BEGIN
      num := scores↑;
      sum := sum + num;
      count := count + 1;
      get (scores)
    END; (* WHILE *)
  avg := sum/count;
  writeln ('the test average is', avg : 7:2)
END.
```

- Notes:
  - Remember that the reset (scores) statement automatically assigns the first component in the file to the buffer variable scores↑.
  - The EOF function is tested immediately after the get procedure to preclude any errors in attempting to retrieve non-existent data.

- UCSD Notes:
  - As with rewrite, UCSD Pascal modifies the reset command to include the actual name in the diskette directory that is to be associated with the file variable named scores. If the diskette file name is 'DATA:TEST1', change the reset (scores) statement to reset (scores, 'DATA:TEST1').
  - All open files must be closed. After processing a file that has been opened by the reset statement, use close (filename) to close the file.

- The alternative form, close (filename, purge) is used to delete an existing file.

- Insert close (scores) after the WHILE statement in the above program.

2. Brooklyn Technical High School wants to post a listing of student grades on the Eleventh Year Mathematics Regents. The listing should include each grade along with the student ID number. Write a program that stores the necessary information in a file and prints a listing of it.

```
PROGRAM Regents (input, output, G);
TYPE
  stugrade =
    RECORD
      idnum, grade : integer
    END;
  gradefile = FILE OF stugrade;
VAR
  choice : integer;
  G : gradefile;
PROCEDURE Createfile;
  VAR
    response : char;
  BEGIN
    rewrite(G);
    REPEAT
      write ('ID number: ');
      readln (G↑.idnum);
      write ('grade: ');
      readln (G↑.grade);
      put(G);
      write ('Another student?  Y or N:');
      readln (response)
    UNTIL response = 'N';
    writeln;
    writeln ('gradefile has been created.')
  END; (* createfile *)
PROCEDURE Printfile;
  BEGIN
    reset(G);
      WHILE NOT EOF(G) DO
        BEGIN
          writeln (G↑.idnum, G↑.grade : 6);
          get(G)
        END (* WHILE *)
  END; (* printfile *)
BEGIN (* regents *)
  writeln ('This program creates and lists a file of student regent
          grades.');
  write ('(1) create file (2) print file (3) exit?  Enter a number: ');
  readln (choice);
  CASE choice OF
    1 : Createfile;
  2 : Printfile;
  3 :
 END (* case *)
END. (* regents *)
```

. Note:

- The CASE statement was used instead of an IF-THEN-ELSE statement so that additional options can be more easily included in the program.
- A warning should be given to the user indicating that the creation of a file erases the old file (e.g. Are you sure?).

. UCSD Note:

- Change the reset and rewrite command to include the name of the diskette file.
- Add close (G, lock) at the end of procedure create file and close(G) at the end of procedure printfile.

<u>Summary:</u>

1. Describe how reset, get and the buffer variable are used to read file G:

'W'	'O'	'L'	'F'	<EOF>

2. How can the statement write (H,74) be written using the get procedure?

3. What happens when we rewrite/reset an existing file?

TOPIC 60

COMBINING FILES

Performance Objectives - Students will be able to:

- append records to a file
- concatenate two files
- write a program that merges two files

Vocabulary: concatenate, merge

Motivation:

- A personnel director would like to append records to a file (named 'EMP86') of employee records. What problems arise?
- If we use rewrite (EMP86), the original file is completely erased.
- If we use reset (EMP86) to get to the end of the file and then read all of the records until eof(EMP86) is true, we are in the wrong mode for writing (i.e. in the read mode).
- What if we tried to concatenate or merge the two files?
- Similar problems would occur.
- How do we append records to a file?

Aim: How can files be combined?

Development:

1. The problem of appending records to a given file can be resolved by making a copy of the original file.

   - Suppose we have the procedure Copyfile (old, new) which leaves us in the write mode at the end of file 'new' -- a duplicate created from file 'old':

```
PROCEDURE Copyfile (VAR old, new : filetype);
BEGIN
  reset (old);
  rewrite (new);
  WHILE NOT eof (old) DO
    BEGIN
      new↑:= old↑;
      put (new);
      get (old)
    END (* while *)
END; (* copyfile *)
```

   - UCSD Note: modify reset/rewrite and add the statement close(old) at the end of the procedure.

- After we invoke procedure copyfile (EMP86,F), records can be easily appended to file F using F↑:= (some new record) and put (F).

2. What problem exists with the procedure Copyfile?

   - The newly created file is no longer called by the original name of EMP86.

3. How can we remedy that?

   - Call procedure copyfile twice:

```
copyfile (EMP86,F);
copyfile (F, EMP86);
(* Append records to file EMP86 *)
```

   - File EMP86 will now contain the newly appended records.

<u>Applications:</u>

1. How can the records in files F and G be concatenated to form a new file H?

```
PROCEDURE Concat (VAR F, G, H : filetype);
  BEGIN
    reset (F);
    reset (G);
    rewrite (H);
    WHILE NOT eof(F) DO
      BEGIN
        H↑:= F↑;
        put(H);
        get(F)
      END;
    WHILE NOT eof(G) DO
      BEGIN
        H↑:= G↑;
        put(H);
        get(G)
      END
END; (* concat *)
```

   - UCSD Note: Modify reset/rewrite and add the statements close (F), close (G), close (H, lock).

2. Given the procedures created, how can we append file G to file F?

   - One method would involve using two procedure calls:
     - concat(F,G,H) and copyfile(H,F).
   - An alternative approach would be to call:
     - copyfile (F,H) and concat(H,G,F).

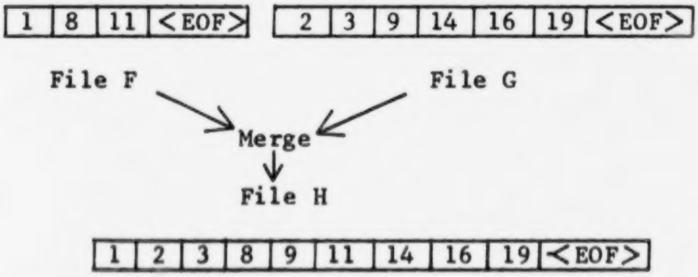
3. Another important type of file manipulation is the sorting of files.

   - When the complete file can be brought into the computer's main memory, a sort such as the bubble sort (or any of the quicker sorts to be discussed in future topics) may be used.

   - However, what can be done to create a sorted file from two other files each of which can be brought into the computer's internal memory alone, but not together?

     - One solution is to sort each file separately and then merge the files to create a new sorted file. We will assume that efficient internal sorting routines exist and concentrate on the process of merging two sorted files.

4. Suppose we have two sorted integer files, F and G, that we would like to merge to form sorted file H:



5. The merge algorithm involves comparing values in the buffer variables F↑ and G↑.

   - The smaller value is put into file H and the corresponding file position is advanced by procedure get.

   - This process is repeated until we reach the end of one of the files. The other file's remaining components are then copied into file H.

   - Program Mergefiles demonstrates this algorithm.

```
PROGRAM Mergefiles (F,G,H);
  VAR
    F,G,H : FILE OF integer;
  BEGIN
    reset (F);
    reset (G);
    rewrite (H);
    WHILE (NOT eof (F)) AND (NOT eof (G)) DO
      IF F↑ <= G↑ THEN
        BEGIN
          H↑ := F↑;
          put (H);
        get (F)
      END (* then *)
    ELSE
      BEGIN
        H↑:= G↑;
        put (H);
        get (G)
      END;(* else *)
  (* transfer remaining elements in non-empty file to H *)
  WHILE NOT eof(F) DO
    BEGIN
      H↑:= F↑;
      put (H);
      get (F)
    END;
  WHILE NOT eof (G) DO
    BEGIN
      H↑:= G↑;
      put (H);
      get (G)
    END
END.
```

. Note: The last two WHILE loops are necessary since either file can be the non-empty one.

. UCSD Note: Modify reset/rewrite and insert statements close (F), close (G), close (H,lock).

Summary:

1. How can the real numbers 2.8, 5.7, 6.2 be appended to file F?

6.1	8.2	3.5	6.4	<EOF>

2. Describe the process of merging two sorted files. Four sorted files.

3. Explain how a file that is too large to bring into the computer's main memory can be sorted. (Break the file into smaller files that can be sorted and merged.)

TOPIC 61

UPDATING FILES

Performance Objectives - Students will be able to:

- describe the need to update files
- update a sequential file
- Show how the UCSD SEEK procedure can be used to update a random access file.

Vocabulary: file update, (UCSD : SEEK)

Motivation:

- The contents of many files have to be constantly changed.
- What changes might be needed in a file of airline reservations? Student records? School personnel?
- The process of altering records in a file is known as updating the file.
- The most common types of file updates are record modification, deletion and insertion.

Aim: How can a file be updated?

Development:

1. To modify a record, the record must first be found in the file, then changed and put back into its previous position.
    - Since we can't read and write a given file at the same time, a new sequential file must be created to store the updated data.
        - By maintaining the large master file (original file) and the smaller transaction file (new information file) in sorted order according to a specified key, the time spent in searching and rewriting a file is shortened.
    - Both files are then sequentially processed to create a new updated master file. These features are illustrated in the next application.

Application:

1. A file of student records has to be updated. Each record contains a student I.D. number, name, high school average, SAT mark and IQ score.
    - Assuming the file is sorted by ID number, write a program to take a similarly sorted collection of data and sequentially modify, delete, and insert records to create a new updated file.

```
PROGRAM Updatefile (input, output, oldfile, newfile);
    (* updates a file of student records *)
CONST
  maxname = 20;
TYPE
  namestring = PACKED ARRAY [1..maxname] OF char;
  student =
    RECORD
      idnum : integer;
      name  : namestring;
      hsavg : real;
      SAT   : 400..1600; (* total *)
      IQ    : integer
    END;
  studentfile = FILE OF student;
VAR
  choice  : char;
  quit,
  found,
  equalid : boolean;  (* flags whether file ID equals requested ID *)
  num     : integer;
  oldfile,
  newfile : studentfile;

PROCEDURE Getmenu (VAR choice : char; VAR quit : boolean);
  (* displays choices in menu *)
BEGIN
  REPEAT
    write ('U)pdate  I)nsert  D)elete  Q)uit)?  Enter choice:');
    readln (choice);
  UNTIL choice IN ['U','I','D','Q'];
  quit := choice = 'Q'
END; (* Getmenu *)

PROCEDURE Readname (VAR N: namestring);
  (* reads name of student *)
CONST
  blank = ' ';
VAR
  I,J : integer;
  ch  : char;
BEGIN
  write ('name:');
  I := 1;
  WHILE (I<= maxname) AND (NOT eoln) DO
    BEGIN
      read (ch);
      N[I] := ch;
      I := I+1
    END;
  FOR J := I TO maxname DO
    N[J] := blank;
  readln
END;
```

```
PROCEDURE Readrecord (VAR S : student; num : integer);
  (* reads a student record *)
BEGIN
  WITH S DO
    BEGIN
      idnum := num;
      Readname (name);
      write ('High school average: ');
      readln (hsavg);
      write ('SAT (total): ');
      readln (SAT);
      write ('IQ score: ');
      readln (IQ)
    END (* WITH *)
END; (* Readrecord *)

PROCEDURE Printrecord (VAR S : student);
  (* prints a student record *)
BEGIN
  WITH S DO
    BEGIN
      writeln ('ID number: ', idnum);
      writeln ('Name: ', name);
      writeln ('High school average: ', hsavg);
      writeln ('SAT (total): ', SAT);
      writeln ('IQ score: ', IQ)
    END (* WITH *)
END; (* Printrecord *)

PROCEDURE Deleterecord (VAR F : studentfile; equalid : boolean);
  (* deletes a record from updated file *)
BEGIN
  IF equalid THEN
    get (F)
  ELSE
    writeln ('Record with ID number was not on file.')
END; (* Deleterecord *)

PROCEDURE Insertrecord  (VAR F : studentfile; num : integer; equalid :
                         boolean);
  (* inserts a record into updated file *)
BEGIN
  IF equalid THEN
    writeln ('Record with ID number is already on file; use update to
            change.')
  ELSE
    BEGIN
      Readrecord (F↑, num);
      put (F)
    END (* else *)
END; (* Insertrecord *)
```

```
PROCEDURE Updaterecord  (VAR S : student; VAR newF, oldF :
                        studentfile; num : integer; equalid : boolean);
  (* updates record in file *)
VAR
  done    : boolean;
  choice : char;
BEGIN
  IF NOT equalid THEN
    writeln ('Record with ID number is not on file.')
  ELSE
    BEGIN
      done := false;
      WHILE NOT done DO
        BEGIN
          REPEAT
            write ('Change  N)ame  A)verage  S)at  I)Q score C)omplete
                  record  P)rint record  or  D)one:');
            readln (choice)
          UNTIL choice IN ['N','A','S','I','C','P','D'];
          done := choice = 'D';
          IF NOT done THEN
            WITH S DO
              CASE choice OF
                'N' : readname (name);
                'A' : BEGIN
                         write ('High school average: ');
                         readln (hsavg)
                      END;
                'S' : BEGIN
                         write ('SAT (total): ');
                         readln (SAT)
                      END;
                'I' : BEGIN
                         write ('IQ score: ');
                         readln (IQ)
                      END;
                'C' : Readrecord (S, num);
                'P' : Printrecord (S)
              END (* case *)
        END; (* while *)
      newF↑:= S;
      put (newF);
      get (oldF)
    END (* else *)
END; (* Updaterecord *)

PROCEDURE Transferfile (VAR F, G : studentfile);
      (* transfers remaining (or all) records from file F to file G *)
BEGIN
  WHILE NOT eof (F) DO
    BEGIN
      G↑:= F↑;
      put (G);
      get (F)
    END (* while *)
END; (* Transferfile *)
```

```
BEGIN (* updatefile *)
  reset (oldfile);
  rewrite (newfile);
  REPEAT
    Getmenu (choice, quit);
    IF NOT quit THEN
      BEGIN
        write ('ID number: ');
        readln (num);
        found := false;
        WHILE (num > oldfile↑.idnum) AND (NOT eof (oldfile)) DO
          BEGIN
            newfile↑:= oldfile↑;
            put(newfile);
            get (oldfile)
          END;
        IF NOT eof (oldfile) THEN
          found := true;
        IF found THEN
          BEGIN
            equalid := oldfile↑.idnum = num;
            CASE choice OF
              'U' : Updaterecord  (oldfile↑, newfile, oldfile, num,
                                   equalid);
              'I' : Insertrecord (newfile, num, equalid);
              'D' : Deleterecord (oldfile, equalid)
            END (* case *)
          END (* if-then *)
      END
  UNTIL quit OR eof (oldfile);
    (* transfer remainder of oldfile to newfile *)
  IF NOT eof (oldfile) THEN
    Transferfile (oldfile, newfile);
      (* rename updated file to oldfile *)
  reset (newfile);
  rewrite (oldfile);
  transferfile (newfile, oldfile)
END. (* updatefile *)
```

2. UCSD Pascal provides a 'seek' procedure that enables a programmer to randomly access any record in a file. SEEK (<filename>, K) causes the file position pointer to advance to record number K in file <filename>. A file of N records has record numbers 0 to N-1.

   . Write a procedure that uses SEEK to change a student's SAT score in a given record of oldfile (that was declared in the previous program).

```
PROCEDURE SATupdate;
  CONST
    maxrecnum = 1000;
  VAR
    recnum : integer;
  BEGIN
```

```
    reset (oldname, '<file diskname>');
    write ('Enter record number or -1: ');
    readln (recnum);
    IF (recnum>= 0) and (recnum<= maxrecnum) THEN
      BEGIN
        seek (oldfile, recnum);
        get (oldfile);
        write ('New SAT score: ');
        readln (oldfile↑.SAT);
        seek (oldfile, recnum);
        put (oldfile);
        write ('Enter record number or -1: ');
        readln (recnum)
      END (* while *)
  END; (* SAT update *)
```

- UCSD Notes:
  - The second SEEK was needed to go back to the original record since the intervening get advanced the file position pointer.
  - Be careful to use record numbers in the proper range. Also avoid using two SEEKS without an intervening get or put.

Summary:

1. What are the most common file update operations?

2. Describe the process by which a file's records can be updated?

TOPIC 62

TEXTFILES

Performance Objectives - Students will be able to:

- describe how textfiles differ from other user-defined files
- explain the special functions and procedures that apply to textfiles
- write a program that copies a textfile

Vocabulary: textfile, text, eoln, writeln, readln

Motivation:

- What would be an appropriate type of file to use in the following circumstances?
  - A student wants to write and modify a book report.
  - A secretary wants to save a business memorandum that can be sent to several corporate executives.
  - A researcher wants to do a textual analysis (e.g. letter and word frequency, line length) of Shakespeare's "Romeo and Juliet".
- In all three situations a file of char values, called a textfile, would be the most beneficial to use.

Aim: What's special about textfiles?

Development:

1. Pascal predeclares a type text to simplify the implementation of textfiles. Its definition

```
TYPE
  text = FILE OF char;
```

is a built-in part of every program. To declare file F as a textfile, we simply write

```
VAR
  F : text;
```

2. A textfile is composed of a sequence of characters usually separated by End Of Line markers (<eoln>).

   - These markers serve to delimit the file into lines of character.
   - Though the control character used for the marker depends on the code used, its value cannot be assigned to a variable of type char.
     - In the ASCII character set, the combination of carriage return and line feed is sometimes used.
   - There are three special operators that specifically relate to textfiles: eoln, readln and writeln.

3. Eoln(<filename>) is a boolean function that indicates whether the <eoln> marker that terminates the current line has been reached.

   - It is true when the character just before the <eoln> marker is read.

     - In this case, the buffer variable contains a blank regardless of the actual control character that is used for the marker.

4. Readln(<filename>, <variable list>) reads values for the variables in the list and then skips to the beginning of the next line.

   - The buffer variable takes on the value of the first character in the new line.

   - Readln without a variable list simply causes a skip to the next line.

5. Writeln(<filename>, <character list>) writes the characters in the list to the textfile together with an <eoln> marker.

   - Writeln without a list causes an <eoln> marker to be generated.

6. Note these differences:

   - All files may use eof, read and write.

     - However, if the file parameter is not of type text, only one component may be included in the variable list.

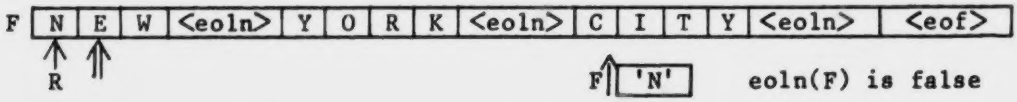
     - Therefore, read(F,X,Y) and write(F,X,Y,Z) are both illegal unless F is a textfile.

   - Only textfiles may use eoln, readln and writeln.

7. Reading a textfile:

   - Let F be a textfile

   - Step 1 -- After reset(F):



     - The symbols R and ⇑ point to two adjacent locations:

       a) ⇑ is the File Position Pointer (FPP). It indicates the location of the next file access, that is, the position of the value that will be put in buffer variable F↑ by the next get(F) procedure.

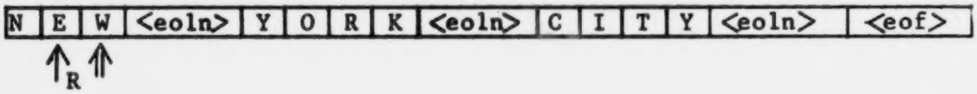
       b) The symbol R points to the next value to be read (which is in the buffer variable F↑).

     - Recall that read(F,X) is equivalent to X := F↑; get(F).

- Therefore, a call to read(F,X) assigns the buffer variable value to the list variable X, and gets the next value (pointed to by FPP) into the buffer variable F↑(for the next read).

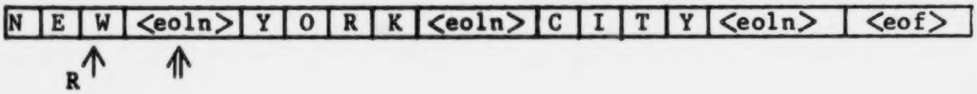
- Since reset(F) automatically assigns the first value ('N') to buffer variable F↑, a call to read(F,X) will result in 'N' being assigned to X and 'E' being assigned to F↑.

. Step 2 -- After the first read(F,X):



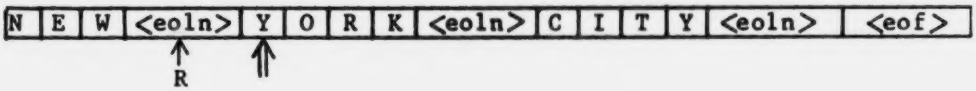
X 'N' F↑ 'E' eoln(F) is false.

. Step 3 -- After the second read(F,X):



X 'E' F↑ 'W' eoln(F) is false.

. Step 4 -- After the third read(F,X):

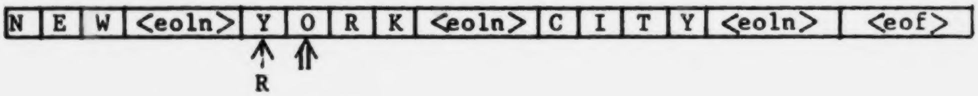


X 'W' F↑ blank eoln(F) is true

- Now that the last character on the line has been read, eoln(F) becomes true.

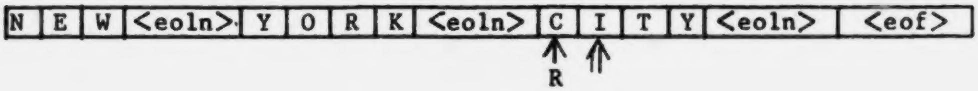
- Another read(F,X) would assign a blank to X and set F↑to 'Y' and eoln(F) to false. To get to the beginning of the next line, without reading the <eoln> as a blank, call readln(F).

. Step 5 -- After readln(F):



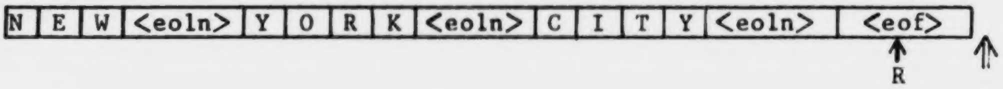
X 'W' F↑ 'Y' eoln(F) is false
(old value)

. Step 6 -- After readln(F,X,Y):



X 'Y' Y 'O' F↑ 'C' eoln(F) is false

. Step 7 -- After another readln(F):



X 'Y' Y 'O' F↑ undefined
(old values)

eof(F) is true
eoln(F) is undefined
(some systems set eoln(F) to
true when eof(F) is true)

8. Copying a file:

. Write a program that copies a textfile while preserving line structure.

. What is wrong with the following program?

```
PROGRAM Copyerror (F,G);
(* (incorrectly) copies textfile F to textfile G *)
VAR
  F,G : text;
  ch : char;
BEGIN
  reset (F);
  rewrite (G);
  WHILE NOT eof (F) DO
    BEGIN
      read (F, ch);
      write (G, ch)
    END (* while *)
END.
```

. Program Copyerror does not preserve line structure since it replaces <eoln> markers with blank characters.

- Remember that reading an <eoln> marker assigns a blank to the list variable; the only way to put an <eoln> marker in a file is to use writeln.

9. The following program signals the end of each line (NOT eoln(F) is false) and uses writeln to copy the <eoln> marker.

```
PROGRAM Copyfile (F,G);
(* copies textfile F to textfile G *)
VAR
  F,G : text;
  ch : char;
BEGIN
  reset (F);
  rewrite (G);
  WHILE NOT eof(F) DO
    BEGIN
      (* copies a line of text *)
```

```
          WHILE NOT eoln(F) DO
            BEGIN
              read (F,ch);
              write (G,ch)
            END;
          readln(F);
          writeln(G)
        END (* while not eof *)
    END.  (* copyfile *)
```

- The inner WHILE loop is exited after the last character in a line is read. Readln skips over the <eoln> marker and takes us to the beginning of the next line. As mentioned previously, writeln copies the <eoln> marker into the new file.

Summary:

1. How do textfiles differ from other standard files? (line structure)

2. Describe the three special operations that apply to textfiles.

Exercise:

1. Modify the copyfile program so that it will replace <eoln> markers by asterisks and blanks by dashes?

TOPIC 63

TEXTFILE APPLICATIONS

Performance Objectives - Students will be able to:

- find a specific record in a textfile which uses end of line markers to separate records
- list three ways textfiles can be used to improve data processing
- explain how integer and real values are read from textfiles
- discuss differences between the predefined input/output files and textfiles

Vocabulary: concatenate, merge sorted files

Motivation:

- Each line of textfile School86 contains information about a specific student. Write a program that efficiently prints the information about a student given his/her line number in the file.
    - It is not necessary to read every character in the file, since the readln procedure can skip over lines.
    - Once the correct line is reached each character in the line can be printed.

Aim: How can textfiles be used to improve the processing of data?

Development:

1. Procedure Findstudent takes as parameters a textfile and line number.

```
PROCEDURE Findstudent (VAR students : text; linenum : integer);
                     (* finds and prints student information *)
VAR
  ch          : char;
  currentline : integer;
BEGIN
  reset (students);
  currentline := 1
  WHILE (currentline <linenum) AND (NOT eof (students)) DO
    BEGIN                            (* find line linenum *)
      readln (students);
      currentline := currentline + 1
    END; (* WHILE *)
  IF NOT eof (students) THEN
    WHILE NOT eoln (students) DO     (* display info on line linenum *)
      BEGIN
        read (students, ch);
        write (ch)
      END
  ELSE
    writeln ('Line number ', linenum, ' does not exist in the file.')
END; (* findstudent *)
```

2. All computers have default devices that carry out standard input and output.

   - In most interactive systems, the usual defaults are to the keyboard for input and the CRT for output.

   - Pascal simplifies operations by associating these devices with the file identifiers input and output.

3. What is the component type of each of these files and how are they used?

   - Both the input and output files are of type text.

     - They have the implicit declaration VAR input, output : text. Because of this, they must never be formally declared.

   - When the input and output files are used, they must be listed as parameters in the program heading.

     - This listing causes the program to automatically generate a reset(input) for the input file and a rewrite(output) for the output file.

4. Why aren't these input and output files listed as parameters in I/O statements?

   - The I/O statements default to these files:

Equivalent Statements

write (ch) ———————— write (output, ch)
read (ch) ———————— read (input, ch)
writeln ———————— writeln (output)
readln ———————— readln (input)
eof ———————— eof (input)
eoln ———————— eoln (input)

5. How can the input and output textfiles be used for reading and writing integer and real values?

   - While it appears that only char values can be read (written) using textfiles, values of any standard type can be read (written) because they are converted from (to) a string of char values.

   - When the computer expects to read an integer, it automatically converts the input characters to an integer value.

     - The conversion involves skipping over leading blanks until the first digit or sign is encountered; all digits are then sequentially read until a non-digit character is found.

   - A similar conversion occurs with real values.

- In many applications, the integer/real conversion of textfiles is inefficient and a file of integer/real is used instead.

6. How are textfiles used for error trapping?

   - Textfiles provide the opportunity to validate input by examining data character by character.

     - The input data is written to a textfile, where a procedure validates it before it is processed by the program.

<u>Applications:</u>

1. Write a procedure that error-checks an unsigned integer value read from a textfile.

```
PROCEDURE Readinteger (VAR F : text; VAR number : integer; VAR error :
                      boolean);
CONST
  blank = ' ';
VAR
  ch : char;
BEGIN
  ch     := blank;
  number := 0;
  error  := false;
  WHILE (NOT eof (F)) AND (ch = blank) DO
    read (F,ch)
  IF NOT eof (F) THEN
    IF ch IN ['0'..'9'] THEN
      REPEAT
        number := 10 * number + (ORD(ch) - ORD('0'));
        read (F,ch)
      UNTIL NOT (ch IN ['0'..'9'])
    ELSE
      error := true
  ELSE
    error := true
END; (* readinteger *)
```

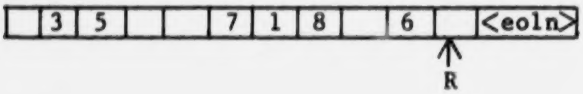
2. Common textfile errors:

   - Given the program segment below and the three indicated integer values input at the keyboard, why is the summing incorrect?

	3	5			7	1	8		6		<eoln>

```
sum := 0;
WHILE NOT eoln DO
  BEGIN
    read (intnum); (* intnum is of type integer *)
    sum := sum + intnum
  END;
writeln ('sum is ', sum);
```

- Answer: After the program reads the third value (6), it gets set to read the next integer value at location R:



- Eoln is false since the character before the <eoln> marker has not been read in yet.

- As soon as the loop executes a fourth time and no integer value is available, the program crashes. (Check your system for specifics; for example, UCSD reads in a value 0.)

- The program would correctly sum the three values if there were no blank after the 6.

. A similar error occurs in the following program segment which attempts to read and display integer values.

  - Remember that an <eoln> marker always precedes the end of a file.

```
WHILE NOT eof DO
  BEGIN
    read (intnum);
    writeln (intnum)
  END;
```

  - This problem of reading past the end of a file can be avoided by reading integer/real values line-by-line.

. How does the compiler deal with the readln statement which can have a multiple number of parameters?

  - It interprets readln (F,VAR1,VAR2,...,VARn), where F is a text file and VARi are variables, as read (F,VAR1); read (F,VAR2); ...; read (F,VARn); readln

  - The read, writeln and write statements are treated in a similar way.

  - readln(F) is equivalent to

```
WHILE NOT eoln (F) DO
  get (F);
get (F);
```

Summary:

1. List three ways in which textfiles can be used to improve the processing of data.

   . Readln skips lines of data

   . Input can be validated

- Real/integer values can be input and output

2. How does the read statement interpret unsigned real values that are input at the keyboard in decimal form?

   - All digits and decimal points after leading blanks are considered to be a part of the real number until a non-digit is encountered.

3. What is a common error in textfile processing?

Exercise:

1. Use the get procedure to write equivalent statements for:

   - readln(F)

   - read(F, intnum)

TOPIC 64

STACKS - A DYNAMIC DATA STRUCTURE

Performance Objectives - Students will be able to:

- explain the differences between dynamic and static data structures and the appropriateness of each
- define the concept of a stack
- explain how the size of a stack can dynamically change
- explain the operations that may be applied to a stack

Vocabulary: dynamic data structure, stack, LIFO, (stack) top, push, pop, underflow

Motivation:

- A parking garage consisting of a single lane contains only one entrance which also serves as the exit. Describe what happens as cars arrive and depart from this garage.

- What data structure can be used to efficiently print a message that indicates:

  - if there is room for arriving cars
  - how many times each car was moved out of the garage to allow other cars to depart

- The data structure which best fits the scheme described is a stack.

- What is a stack and how does it differ from other structures studied so far?

Aim: When and how are stacks used in programs?

Development:

1. A stack is a linearly ordered collection of homogeneous items into which new items may be inserted or removed from one and only one end.

   - This end is called the top of the stack.

2. How does a stack differ from an array?

   - A stack is similar to an array in that items are homogeneous and ordered.

   - It is different from an array in the following manners:

     - The size of an array is static (fixed) throughout a program whereas the size of a stack is dynamic (fluctuates) as the program is executed.

  - The size of a static data structure is determined in the declaration of the program. The size of a dynamic data structure is not predeclared and changes during execution of the program.

  - Data can be inserted or deleted from anyplace in an array whereas data can be added and retrieved only from the top of a stack.

  - Sometimes stacks are called LIFO (Last In First Out) structures -- a term in accounting, where, for inventory purposes, the last item purchased is assumed to be the first item that is sold. Similarly, in a stack, the last item inserted is the first to be removed.

3. While the cars in the parking garage problem could be represented by an array, what problems might arise from such a representation?

   - The maximum number of cars the garage can accommodate must be determined at the outset because the array size must be declared.

   - If substantially fewer than the maximum use the garage, there is a waste of memory space.

   - Since cars can be added or removed from the garage only from the end of the lane, analogous deletions from an array will leave a gap in the array which must then be indicated.

4. When is it appropriate to use the structure of a stack?

   - Stacks may be used in any situation that calls for a last in, first out (LIFO) pattern. For example:

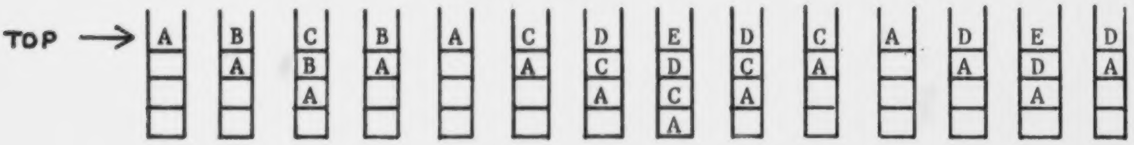
     - the order of cafeteria trays as they are used and replaced,

     - the sequence in which the round trips would be used from a series of round trip tickets,

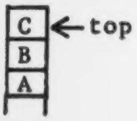
     - the order in which procedures (or subroutines) are called and returned.

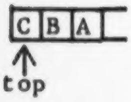
5. Suppose that cars A, B, and C arrive at the garage in that order. Car B departs and cars D and E arrive. Cars C and E then depart in that order.

   - Draw a diagram depicting every state that the stack, which represents the cars in the garage, assumes.

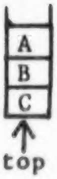


   - Note that a stack may be visualized in any position:

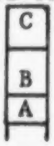
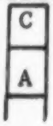




or



as long as the top is indicated.

- In most examples the only element of a stack that can be viewed is its top element. Thus  and  appear to be the same since both have a top element of C.

- The only way to determine their difference is to remove all the elements from each and compare them individually.

6. The operation of adding an element to a stack is called a push.

   - Adding element I to stack S can be denoted as push(S,I). A stack is sometimes called a pushdown list.

7. Removing the top element from a stack can be denoted as pop (S).

   - This will be expressed as a function where I := pop(S) removes the top element from stack S and assigns it to I.

8. The example in number 5 above can be denoted by the following sequence of calls: push(S,A), push(S,B), push(S,C), I := pop(S), J := pop(S), push(S,I), push(S,D), push(S,E), ...

9. Can you always perform a pop or a push on a stack?

   - If a stack is empty, no pop can be performed. An attempt to do so results in an underflow condition.

   - In theory a stack has no limit on its size. However, the limited size of computer memory and the particular implementation being used will restrict the size of the stack.

Summary:

1. What is the difference between a dynamic and a static data structure?

2. What is a stack and when would it be used?

3. Describe the operations that can be performed on a stack.

Exercises:

1. Name two static data structures in Pascal.

2. Explain how the order of operations in the computation of the following expression conforms to the structure of a stack: {7 - [3 + (2 * 6)]}

3. Diagram the states of stack S as the following operations are performed. Assume S is empty at the outset:

   Push(S,2), Push(S,3), Push(S,4), I := Pop(S), Push(S,5), Push(S,I).

   - What element is at the top of stack S at the end of the above sequence of procedure calls?

4. Assume that at the end of the sequence of procedure calls in example (3), S was popped 3 times and each element as it was popped was pushed onto stack T.

   - What element would be at the top of S?

   - What element would be at the top of T?

TOPIC 65

IMPLEMENTING A STACK

Performance Objectives - Students will be able to:

- describe how a stack may be implemented using the structure of a record
- discuss the problems of stack underflow and overflow
- write subprograms that implement the pop and push operations, and functions to determine if the stack is empty or full

Vocabulary: overflow

Motivation:

- In order to create a data structure to conform to the characteristics of the motivation problem in TOPIC 64, a programmer defines the following variables:

```
cars :  ('A'..'H');
lane :  ARRAY[1..8] OF cars;
```

- If he is then provided with a time schedule of arriving and departing cars, what additional designation must be make before he can write a program for the motivation problem?

- He must designate the location of the entrance/exit to the garage.

Aim: How do you implement a stack?

Development:

1. A major difference between a stack and an array is in the way its elements are accessed.

   - In a stack, the only way the elements can be added or deleted is from its top.

   - Therefore it is always necessary to know the location of the top of the stack.

2. What data structure or structures already studied would best be suited to represent a stack?

   - Two ingredients are necessary to represent a stack:

     - the ordered elements, which can be represented by an array, and

     - the location of the top, which can be indicated by an integer that corresponds to the index of the element at the top.

   - Since both ingredients are represented by different types, a record containing these two fields is an appropriate structure with which to declare a stack.

3. In general, a stack can be declared in the following way:

```
TYPE
  Stack =
    RECORD
      top : integer;
      item : ARRAY [1..N] OF elements
    END;
VAR
  S : stack;
```

   - In this declaration, elements can be any standard or user-defined data type and it is assumed that the stack will never exceed N elements.

4. What problems will be encountered if we designate index one to always be the top?

   - Each time an element is deleted or added to the stack, all the elements in the array will have to be shifted.

     - It is easier to change the value of the integer representing the stack top.

     - When a stack is popped, the value of S.top must be decreased by 1.

     - When an element is pushed onto the stack, S.top must be increased by 1 and the new element assigned to S.item[S.top].

5. Can pops and pushes always be performed?

   - An attempt to pop an empty stack results in underflow and can be avoided by first utilizing a boolean function which determines if the stack is empty.

     - Function Empty(S) can be defined to be true when S.top = 0.

     - Note that the use of a Function Empty (instead of just testing if S.Top = 0) will be more meaningful later when non-array stack implementations are introduced.

   - Theoretically, an element can always be pushed onto a stack; however, there are physical limitations. When an array is used to represent a stack, the size of the stack cannot exceed that of the array.

     - An attempt to push an element onto a full stack, which would result in overflow, can be prevented by utilizing a boolean Function Full(S). This function can be defined to be true when S.top = N where N is the number of elements in the array.

     - Another way to correct an overflow condition is to change the value of the constant N in the declaration and rerun the program.

6. The following program simulates a stack:

```
PROGRAM Stacklike;
CONST
  N = 8;
TYPE
  choicetype = 1..4;
  elements = integer;
  stack =
    RECORD
      top : 0..N;
      item : ARRAY [1..N] OF elements
    End; (* Stack *)
VAR
  garage : stack;
  Y : elements;
  Z : choicetype;

FUNCTION Empty (X : stack) : boolean;
BEGIN
  IF X.top = 0 THEN
    empty := TRUE
  ELSE
    empty := FALSE
END;

FUNCTION Full (X : stack) : boolean;
BEGIN
  IF X.top = N THEN
    full := TRUE
  ELSE
    full := FALSE
END;

FUNCTION Pop (VAR X : stack) : elements;
BEGIN
  IF Empty(X) THEN
    writeln ('Stack underflow')
  ELSE
    BEGIN
      Pop := X.item[X.top];
      X.top := X.top - 1
    END
END;

PROCEDURE Push (VAR X : stack; E : elements);
BEGIN
  IF Full(X) THEN
    writeln ('Stack overflow')
  ELSE
    BEGIN
      writeln ('PUSH');
      X.top := X.top + 1;
      X.item[X.top] := E
    END
END;
```

```
PROCEDURE Initialize (VAR X : stack);
BEGIN
  X.top := 0
END;

PROCEDURE Print (X : stack);
VAR
  I : integer;
BEGIN
  IF Empty(X) THEN
    writeln ('EMPTY')
  ELSE
    BEGIN
      write ('Stack = ');
      FOR I := X.top DOWNTO 1 DO
        writeln (X.item[I])
    END;
  writeln
END;

PROCEDURE Menu (VAR choice : choicetype);
BEGIN
  writeln ('1   pop');
  writeln ('2   push');
  writeln ('3   initialize');
  writeln ('4   end');
  writeln ('Type your choice: ');
  readln (choice)
END;

BEGIN (* main program *)
  Initialize (garage);
  Menu (Z);
  WHILE Z<>4 DO
    BEGIN
      IF Z = 1 THEN
        BEGIN
          Y := pop(garage);
          Print (garage)
        END
      ELSE
        IF Z = 2 THEN
          BEGIN
            writeln ('Give the element: ');
            readln (Y);
            push (garage, Y);
            Print (garage)
          END
        ELSE
          Initialize (garage);
      Menu (Z)
  END (* WHILE *)
END.
```

Summary:

1. How can a stack be implemented using the structure of a record?

2. What precautions can be taken to prevent underflow or overflow errors from occuring?

3. Describe a procedure for pushing an element onto a non-full stack using the stack implementation presented in this lesson.

Exercises:

1. Rewrite the routines of this lesson defining Function Empty to be true if and only if S.Top = 1 and Function Full to be true if and only if S.Top = N - 1.

2. Using the routines of this lesson write procedures that will:

   - Set the top of the stack to the second element from the top, leaving the stack without its first two elements.

   - Reverse the elements in a stack, making the bottom element the top and vice-versa.

TOPIC 66

USING A STACK FOR ERROR CHECKING

Performance Objectives - Students will be able to:

- describe how a stack can be used to verify whether parentheses are correctly nested in an expression
- explain a pseudo-Pascal coded algorithm that implements the above check for proper nesting in an expression

Motivation:

- Which of the following mathematical expressions uses parentheses and nested parentheses correctly? Explain.
- (A + B - (C+ D)
- A + B) - (C + (D - E)
- (A + (B - C * (D + E) + F) - G)
- Elicit that only the third of the above examples is correct.

Aim: How can we use a stack to check if an expression has correctly nested parentheses?

Development:

1. What rules govern the use of parentheses to phrase mathematical expressions?
   - There must be an equal number of left and right parentheses (openers and enders).
     - The first example above fails to satisfy this requirement.
   - Every right parenthesis must be preceded by a left parenthesis. The last parenthesis to be opened must be the first to be closed.
     - The second example above fails to satisfy this requirement.
2. How can the computer determine if a string is valid according to these rules?
   - One method is to read the string from left to right and keep count of the number of left and right parentheses. The expression is valid if:
     - At every point the total number of left parentheses minus the total number of right parentheses is never negative, and
     - The result equals zero at the end of the expression.
   - What does a negative count indicate? A final count of zero?

- Have the students walk through this procedure using the three expressions in the motivation problem.

3. In complicated expressions, such as the third example, several types of scopes are sometimes used to clarify the phrasing.

    - For example, it could be expressed {A + [B - C * (D + E) + F] - G}.

    - When such a system is used, each type of scope ender must have a matching scope opener which precedes it.

4. How can we test for the validity of expressions which use several types of scopes?

    - It is not sufficient merely to keep count of the number of openers and enders, as in the problem which deals only with parentheses.

    - It is also necessary to keep track of the different types of scopes opened and insure that each is closed properly, that is, in the proper order.

        - For example, is the following expression valid?

          [A + (B + C] * D).

        - Although each different scope ender is preceded by a scope opener, this statement is not valid since the last scope opened is not the first closed.

        - The determination of the pattern of last opened/first closed to test the validity of these expressions should suggest the use of a stack.

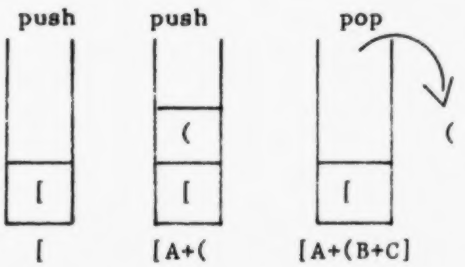
5. How do we implement a program for reading different types of scopes?

    - The expression will be read from left to right. Each time a scope opener is encountered it will be pushed onto a stack. This stack is to be examined whenever a scope ender is encountered.

        - If the stack is empty, the scope ender does not have a matching opener and the expression is not valid.

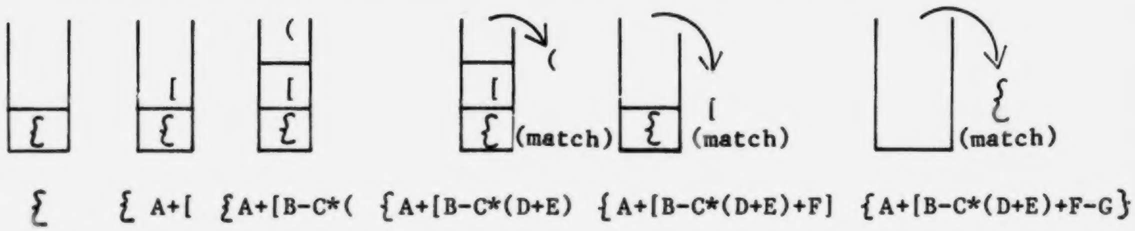
        - If the stack is not empty, it is popped and the popped opener is compared to the ender. If it matches in type, the process is continued.

6. In the example above the stack would develop as follows



The popped "(" is compared to the ender "]". It does not match. Thus the expression is invalid.

- Have the class draw the stacks that would be constructed as the following string is examined: {A + [B - C * (D + E) + F] - G}



- For the expression to be valid, the stack must be empty at the end of the string.

6. The following algorithm outlines the above procedure:

```
valid := true;
initialize (S);
WHILE (NOT eoln) AND valid DO        (* we have not read the entire
                                     expression *)
  BEGIN
    read (ch);   (* the next symbol *)
    IF ch IN ['(','[','{'] THEN
      push (S, ch);
    IF ch IN [')',']','}'] THEN
      IF Empty(S) THEN
        valid := false
      ELSE
        BEGIN
          I := pop(S);
          IF ((I = '(') AND (ch<>')')) OR ((I = '[') AND (ch<>
          ']')) OR ((I = '{') AND (ch<>'}')) THEN
            valid := false
        END (* else *)
  END; (* while *)
IF NOT Empty(S) THEN
  valid := false;
IF valid THEN
  writeln ('The string is valid.')
ELSE
  writeln ('The string is invalid.');
```

Summary:

1. Describe an algorithm that can be used to verify whether parentheses are correctly nested in an expression.

2. Describe a method that can be used to verify the validity of an expression if scopes of different types are used within it.

3. What does each element on the stack in the algorithm developed above represent?

4. When will this stack be popped?

Exercises:

1. Write a program that determines the validity of an expression which uses only parentheses and nested parentheses for its phrasing.

    . Use the method outlined in (2) of the development.

2. Write a procedure that will determine if I is a matching opener for the symbol ch in the algorithm outlined in the development.

3. Simulate the action of the algorithm to examine the strings representing the expressions which follow. Exhibit the contents of the stack at each stage. Indicate from the simulation if the expression is valid.

    . (A + B] - C)

    . {(A + B) - [C - (D - E)}]

    . {([X + Y])

    . {X + (Y/[H + I] + J) * K}

TOPIC 67

POSTFIX EXPRESSIONS

Performance Objectives - Students will be able to:

- evaluate postfix and prefix expressions
- change expressions from postfix or prefix to infix form and vice-versa
- describe an algorithm that uses stacks to evaluate postfix expressions

Motivation:

- The following command sequence is executed on the initially empty stack, S:

```
push(S,3); push(S,5); push(S,7);
A := pop(S); B := pop(S);
C := A + B; push(S,C);
D := pop(S); E := pop(S);
X := D * E; push(S,X);
writeln ('Value is ', X);
```

- What is the value of X after these instructions are executed? (X = 36.)

- What is the state of the stack S after these instructions are executed? (Stack S contains the single element 36.)

- Write a mathematical expression indicating the operations that were performed in the above program segment. [Answer: (7 + 5) * 3]

Aim: How can stacks be used to evaluate postfix expressions?

Development:

1. Note that (A + B) * D is different from A + B * D (or its equivalent A + [B * D]).

    - The notation used here which inserts the operation + or * between the operands A, B and D is called infix notation.

    - Use of this notation for evaluation of arithmetic expressions presupposes knowledge of the rules of "operational precedence", that is, multiplication and division are performed before addition and/or subtraction unless parentheses indicate otherwise.

    - Writing complicated arithmetic expressions in infix notation requires frequent use of parentheses to indicate the order in which the operations are to be performed. Such frequent use makes writing a program to evaluate them complicated and awkward.

    - Two other notations for expressing indicated operations in arithmetic expressions exist. They are prefix and postfix notation, both of which eliminate the need for parentheses and simplify the evaluation process.

2. In postfix notation, each operator (+, -, *, /) in the expression operates on the two previous operands (A, B, C, etc).

   - For example, A B + (in postfix notation) is equivalent to A + B (in infix notation).

   - A previous operand may be the result of an operation itself.

     - The postfix expression A B C + - indicates that B and C are to be added and their sum is to be subtracted from A; that is, it is equivalent to A - (B + C) in infix notation.

   - An expression in postfix notation is not valid if an operator is not preceded by two operands.

3. In prefix notation each operator in the expression precedes the two operands to which it refers.

   - For example, + A B in prefix corresponds to A + B in infix notation, and - A + B C in prefix corresponds to A - (B + C) in infix notation.

   - Evaluating an expression written in prefix notation requires no parentheses, but the operation cannot be performed unless an operator is followed by two operands.

   - To convert from infix to postfix (prefix):

     - Translate operations in order of highest precedence;

     - Replace infix expression by the two operands followed (preceded) by the operator; and,

     - Once a portion of the expression has been translated, it should be treated as a single unit.

     - Remove parentheses.

   - For example, change (A + B) - C * D to postfix:

     (A B +) - C * D
     (A B +) - (C D *)
     A B + C D * -

4. Convert the following infix expressions to their postfix and prefix equivalents:

	infix	postfix	prefix
1.	A - B	A B -	- A B
2.	A * B / C	A B * C /	/ * A B C
3.	(A + B) / (C + D)	A B + C D + /	/ + A B + C D
4.	A-B*C+D-E/F*(G+H)	ABC*-D+EF/GH+*-	-+-A*BCD*/EF+GH

- When converting from infix to postfix, the operands appear in the same order. Therefore, it is necessary to know the correct order of operations to properly insert the operators.

- For example, in number 3 above, you must first evaluate A + B and C + D before dividing the results. AB + and CD + are then treated as single operands to be divided.

5. How can we utilize postfix notation in planning a program to evaluate arithmetic expressions?

   - Evaluate the following postfix-notated expressions.

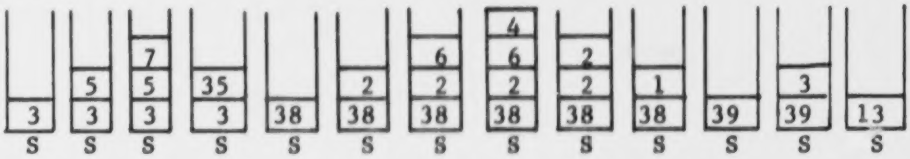
7 5 + 3 *	(7 and 5 are added to produce 12, which is then multiplied by 3 to produce 36)
7 5 3 * +	(5 and 3 are multiplied to produce 15, which is then added to 7 to produce 22.)

   - When writing a program to evaluate postfix expressions, an arithmetic expression can be implemented as an input line in the form of a string of operands (digits) and operator symbols.

     - Whenever an operand is encountered while reading the string, it can be pushed onto a stack.

     - When an operator is encountered, the stack could be popped twice, the operation performed and the result pushed onto the stack to be available, if needed, as an operand for the next operator.

6. Note each stage of stack S as the algorithm described above is applied to the following string of digits: 3 5 7 * + 2 6 4 - / + 3 /



7. Since the symbols are read as characters, we must convert operand characters to numbers, and operator characters to operations.

   - Ord(ch) - Ord('0') accomplishes the first of these conversions.

   - The class should write a function to perform the second.

```
FUNCTION Evaluate (symb: char; num1, num2: real) : real;
BEGIN
  IF symb IN ['+','*','-','/'] THEN
    CASE symb OF
      '+' :  evaluate := num1 + num2;
      '*' :  evaluate := num1 * num2;
      '-' :  evaluate := num1 - num2;
      '/' :  evaluate := num1 / num2
```

```
   END (* Case *)
  ELSE
   writeln ('Illegal evaluation.')
 END; (* Evaluate *)
```

8. The following function (which evaluates a postfix string) assumes input to be a string of operands and operator characters. The operands are single non-negative digits. The string has 81 positions (0-80) and terminates with a blank. Evaluation continues until a blank is encountered.

```
FUNCTION Val (expression : string) : real;
CONST
  maxstack = 80;    (* 80 is the max # of columns on an input line *)
TYPE
  stack =
    RECORD
      num : ARRAY [1..maxstack] OF real;
      top : 0..maxstack
    END; (* record *)
VAR
  S    : stack;
  num1,
  num2,
  X    : real;
  ch   : char;
  I    : 0..maxstack;
BEGIN
  S.top := 0;
  I     := 0;
  ch    := expression[I]   (* I denotes the present position in the
                              expression and ch is the present symbol *);
  WHILE (ch<>' ') AND (I < 80) DO  (* scan until a blank is
                                      encountered *);
    BEGIN
      IF (Ord(ch)>= Ord('0')) AND (Ord(ch)<= Ord('9')) THEN
        BEGIN       (* operand is a one-digit number *)
          X := Ord(ch) - Ord('0');
          push (S,X)
        END
      ELSE
        BEGIN
          num2 := pop(S);
          num1 := pop(S);
          X := Evaluate(ch, num1, num2);
          push (S, X)
        END;
      I := I + 1;
      ch := expression[I]
    END; (* while *)
    val := pop(S)
END;
```

Summary:

1. Describe the differences between infix, postfix and prefix notations.
2. When is a postfix-notated expression invalid?
3. How do you change an expression from infix to postfix notation?
4. Describe an algorithm that reads and evaluates a string of operands and operators written in postfix notation.

Exercise:

1. Convert each of the following infix expressions to postfix and prefix notation:
   - A*B+C
   - A*(B+C)
   - (H+J)-(K+L)
   - ((A+B)-(C+D)-E)*F
   - (E+F)*(G/H+I)/(J-K)
2. Are the following expressions valid postfix notations? If not, indicate why.
   - 2 5 7 + *
   - 3 7 * 2 +
   - 4 2 - 3 + 9 7 6 - + *
3. Are the following expressions valid prefix notations? If not, indicate why.
   - + 2 5 - 3
   - * - 7 1 6
   - + 2 - 7 6
4. Modify FUNCTION Evaluate to include exponentiation as a fifth operator.
5. Modify FUNCTION Val to test for errors of too many operands or too many operators in the input string. Print out the appropriate error message.
6. Rewrite FUNCTION Val to accommodate negative digits. To avoid confusion with the subtraction symbol use a symbol other than '-' in the input string to represent negative numbers (e.g. %).

TOPIC 68

THE STRUCTURE OF A QUEUE

Performance Objectives - Students will be able to:

- define the concept of a queue
- explain operations that may be applied to queues
- discuss how a queue can be represented in Pascal

Vocabulary: queue, front, rear, insert and remove operations

Motivation:

- A parking garage contains a single lane which can hold five cars. There is a single entrance at one end of the lane and a single exit at the other end. Describe what happens as cars arrive and depart from the garage. How can we simulate this situation in Pascal?

Aim: What is a queue?

Development:

1. A queue is a linear structure in which elements are entered at one end (rear) and removed from the other (front). The information in a queue is homogeneous and ordered.
   - Such a structure is called FIFO for data which is processed according to a first in/first out rule.
2. When is a queue used?
   - It is appropriate to use the structure of a queue whenever data is to be processed on a first in/first out basis such as people waiting on lines at movie theaters, banks or supermarkets.
3. How can we represent a queue in Pascal?
   - An array can hold the elements, and two variables, front and rear, can indicate the positions in the array of the first and last elements of the queue.
   - The garage problem above may be declared as follows:

```
CONST
  maxnum = 5;
TYPE
  queue =
    RECORD
      items : ARRAY [1..maxnum] OF 'A'..'Z';
```

```
          front,
          rear  :  0..maxnum
        END;
    VAR
      Q : queue;
```

4. What operations do we need to be able to perform on a queue?

   . We need procedures to insert and delete items.

     - Note that an attempt to remove an item from an empty queue results in an underflow.

     - While we can always theoretically insert items onto a queue, there are physical limits.

   . Thus, we shall need the following:

     - Insert (Q, X) as a procedure which inserts item X at the rear of Q.

     - Remove (Q, X) as a procedure which deletes the front item from Q and sets X to its value.

     - Empty (Q) as a boolean function which returns false or true depending on whether or not Q contains any elements.

     - Full (Q) as a boolean function similar to Empty(Q).

5. Under the assumption that Q.rear = 0 and Q.front = 1 represents an empty queue, depict the following set of instructions:

   Insert(Q, 'A'), Insert(Q, 'B'), Insert(Q, 'C'), Remove(Q, X),
   Insert(Q, 'D'), Insert(Q, 'E').

	1	2	3	4	5	
Insert(Q,'A') -->	A					Front = 1 Rear = 1
Insert(Q,'B') -->	A	B				Front = 1 Rear = 2
Insert(Q,'C') ---->	A	B	C			Front = 1 Rear = 3
Remove(Q,X) ----->		B	C			Front = 2 Rear = 3
Insert(Q,'D') ---->		B	C	D		Front = 2 Rear = 4
Insert(Q,'E') ---->		B	C	D	E	Front = 2 Rear = 5

- What happens if we try to Insert (Q,'F') at this point?

- Note that we cannot insert at the rear even though Q is not full (Q.items[1] is empty).

6. There are several ways in which we can avoid such situations and obtain maximum utilization from our array.

   - In one method, each time Remove(Q, X) is called all values in the array would be shifted so that the front of the queue is always in the first position of the array. For long queues this method is very inefficient.

   - A second method is to wait until an element is added to the last location in the array, Q.items[maxnum]. At that time you can move all the elements down so that the first element is in location Q.items[1] and the other elements are in the following successive locations.

   - Another method involves conceptualizing the queue as a ring in which N contains the maximum index of the array, 'length' contains the number of elements in the queue at any given time, 'front' contains the index of the element at the front of the queue and 'rear' contains the index of the element at the rear of the queue (front and rear serve as pointers).

   - Using this scheme the following declaration can be used:

```
CONST
  N = 5;
TYPE
  elements = integer;
  queue =
    RECORD
      items : ARRAY [1..N] OF elements;
      front, rear, length : 0..N
    END; (* Queue *)
VAR
  Q  : queue;
```

   - If Q.rear = N, an integer can still be inserted into Q (provided Q.front > 1) by inserting it at Q.items[1] and changing the value of Q.rear to 1.

7. How can we tell if a queue is empty using the above declaration?

   - We can determine if Q is empty by checking for Q.length = 0. Without Q.length we would not be able to determine whether or not Q is empty.

   - Consider the following situations: First, if Q.front = 4 and Q.rear = 2 and we insert another element into the queue to fill it, we will have Q.front = 4 and Q.rear = 3. Next, if we have only one element in the queue with Q.front = 4 and Q.rear = 4 and we remove that element, we will have Q.front = 4 and Q.rear = 3. Therefore, when Q.front = 4 and Q.rear = 3, we cannot determine whether Q is full or empty. We must use Q.length to make the distinction.

8. How can we implement a queue without having to keep track of its length?

   - We can use the previous implementation as long as we never allow the array to become completely full. We must consider our queue to be full when there is only one free location in the array.

Summary:

1. What is a queue?

2. When should a queue be used in programming?

3. How may a queue be represented using arrays?

4. What operations for queues will we need to develop?

Exercises:

1. Give some examples of queues in the real world and state the meaning of the Remove and Insert routines that correspond to them.

2. Describe how a shop owner keeps track of the value of his inventory when he uses a queue to store each item's purchase price.

3. Assume items X1, X2, X3, X4, X5 and X6 are pushed, in that order, onto an initially empty stack S. S is then popped four times. As each Xi is popped it is inserted onto an initially empty queue Q. An element is then deleted from Q.

   - Draw a diagram which represents S and Q at the conclusion of the described sequence of events.

   - What element is now on top of S?

   - What element was deleted from Q?

   - What element is the next to be deleted from Q?

4. Show how a sequence of insertions and removals from a queue represented by a linear array can cause the occurrence of an overflow when an attempt is made to insert an element into an empty queue.

5. Write the routines Remove(Q,X), Insert(Q,X), and Empty(Q) assuming that a queue is represented by a linear array in which front is always the first element. Note: Initially, Q.rear is set to 0 and Q.front is set to 1.

TOPIC 69

IMPLEMENTATION OF QUEUE OPERATIONS

Performance Objectives - Students will be able to:

- discuss problems related to programming various routines using the ring representation of a queue
- explain how the queue operations of remove, insert, empty, full, print and next can be implemented in Pascal.

Motivation:

- List all the procedures and functions that will be needed in a program designed to simulate the queue of the parking garage in Topic 68.

Aim: How can we implement queue operations?

Development:

1. In addition to the procedures of Insert and Remove and the functions of Full and Empty discussed in Topic 68, it would be desirable to develop procedures and functions that would do the following:

   - Initialize the queue to an empty queue.

   - Print the state of the queue at a given moment.

   - Indicate the front element in the queue at a given moment.

2. In implementing the insert procedure using the ring model of a queue described in Topic 68, what special situations should we be aware of?

   - Discuss the need to first consider the case in which the queue is full. A warning message should be included to avoid overflow.

   - In other cases the value for R, the index of the rear item, must be changed after an insertion has occurred. Consider the case in which the queue is not full but the last item in the queue is in the Nth cell of the array. Elicit that in such a case the item should be inserted in the first cell of the array (giving R the value 1).

   - In all other situations the item to be inserted is inserted in the cell of the array with the index of one more than the previous rear item (R = R + 1). Also mention the need to increase length, the number of items in the queue, after an insertion has taken place.

3. In implementing the remove procedure, what special situations should be considered?

   - A warning should be given if the remove procedure is attempted on an empty queue.

- In other cases the front index, F, must be changed each time an item is removed from the queue unless the queue only contains a single element.
  - Assuming there is more than a single element in the queue, if the front item is in the Nth position of the array, it should be removed and the new front should become 1.
- In all other cases the new front F becomes one more than the previous front (F = F + 1).
- Do not forget to decrease the number of elements in the queue by one (length = length - 1) each time a remove is performed.

4. What difficulties do you foresee in implementing the print procedure?

   - Indicate that in the ring model there is no difficulty if F<= R. This, however, is not always the case.
   - Devise an algorithm for printing the elements in the queue if F > R.

5. Using the ring model of a queue described in Topic 68, the following program implements the routines mentioned above:

```
PROGRAM Queuesim; (* Simulates a queue *)
CONST
  N = 5;
TYPE
  choicetype = 1..4;
  elements = integer;
  queue =
    RECORD
      F, R, length : 0..N;
      items : ARRAY [1..N] OF elements
    END; (* Queue *)
VAR
  choice : choicetype;
  Q : queue;
  I, W, Y, Z : elements;

PROCEDURE Initialize (VAR X : queue);
BEGIN
  X.F := 1;
  X.R := 0;
  X.length := 0
END;(* Initialize *)

FUNCTION Empty (X : queue) : boolean;
  (* determines if queue is empty *)
BEGIN
  IF X.length = 0 THEN
    empty := true
  ELSE
    empty := false
END; (* Empty *)

FUNCTION Full (VAR X : queue): boolean;
BEGIN
  IF X.length = N THEN
    full := true
  ELSE
    full := false
END; (* Full *)

PROCEDURE Insert (VAR X : queue; E : elements);
BEGIN
  IF Full(X) THEN
    writeln ('Queue overflow')
  ELSE
    BEGIN
      IF X.R := N THEN
        X.R := 1
      ELSE
        X.R := X.R + 1;
      X.items[X.R] := E;
      X.length := X.length + 1
    END
END; (* Insert *)

PROCEDURE Remove (VAR X : queue; VAR E : elements);
BEGIN
  IF Empty(X) THEN
    writeln ('Queue Underflow')
  ELSE
    BEGIN
      E := X.items[X.F];
      IF X.F = N THEN
        X.F := 1
      ELSE
        X.F := X.F + 1
    END;
  X.length := X.length - 1
END; (* Remove *)

PROCEDURE Print (X: queue);
BEGIN
  IF Empty(X) THEN
    writeln ('Queue is empty')
  ELSE
    BEGIN
      write (' Queue = ');
      IF X.F<= X.R THEN
        FOR I := X.F TO X.R DO
          write (X.items[I], ' ')
      ELSE
```

```
        BEGIN
          FOR I := X.F TO N DO
            write (X.items[I], ' ');
          FOR I := 1 TO X.R DO
            write (X.items[I], ' ')
        END;
      writeln
    END;
  writeln
END; (* Print *)

PROCEDURE Menu (VAR choice : choicetype);
BEGIN
  writeln ('1     Remove');
  writeln ('2     Insert');
  writeln ('3     Initialize');
  writeln ('4     End');
  writeln ('Type your choice');
  readln (choice)
END;

BEGIN (* Main Program *)
  Initialize (Q);
  Menu (Z);
  WHILE Z<>4 DO
    BEGIN
      IF Z = 1 THEN
        BEGIN
          Remove(Q, W);
          Print(Q)
        END
      ELSE
        IF Z = 2 THEN
          BEGIN
            writeln ('Give the element');
            readln(W);
            Insert(Q, W);
            Print(Q)
          END
        ELSE
          Initialize (Q);
      Menu (Z)
    END(* While *)
END.
```

Summary:

1. What operations need to be implemented to reflect the changes in a queue?

2. Describe the difficulties that arise in implementing remove, insert, and print routines when a queue is conceived as a ring.

Exercises:

1. Using the routines developed in this lesson, write a program to add the numbers in a queue at any given state.

2. Write a program to find the largest element of a queue, leaving the queue unchanged. What additional data structure is needed?

3. Write a procedure that will determine the value of length (the number of elements in the queue) from the values of F and R (the index of the first and last elements in the queue).

TOPIC 70

LINEAR LINKED LISTS

Performance Objectives - Students will be able to:

- cite the major disadvantages of using arrays to represent stacks and queues
- define a linked list and related terms
- use algorithmic notation to demonstrate how elements are inserted or removed from a linked list

Vocabulary: linked list, pointer, information field, next address field, node, NIL, getnode operation, freenode operation, algorithmic notation

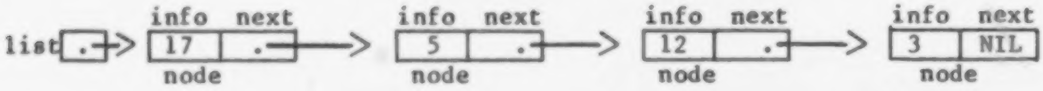
Motivation:

- When using arrays to implement stacks and queues, there are two major disadvantages.
  - One is the possibility of overflow. Even if we dimension our arrays to handle the largest number of expected elements, we could still exceed this maximum and get an overflow error.
  - Another disadvantage is the waste of storage space. Dimensioning an array causes that amount of storage space to be reserved throughout the program's execution. That fixed amount remains allocated to the stack or queue even when the structure contains only a few or no elements at all.
- The next few lessons will cover the abstract concept of linked lists and how they are used with certain Pascal features to overcome these problems.

Aim: What is a linear linked list?

Development:

1. When we use arrays to implement stacks and queues, we are using a sequential representation. That is, the elements of the stack or queue are implicitly ordered by their sequential order in the array.
   - In a linear linked list, the elements are ordered explicitly rather than implicitly. That is, each element contains within itself the address of the next element. Such a data structure is pictured below.



2. The diagram represents a linear linked list, sometimes referred to as a singly linked list or simply, a linked list. Each item in the list is called a node and contains two fields: an information field and a next address field.

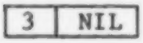
- The information field contains the actual element on the list. The next address field contains the address of the next node in the list. An address of a node is known as a pointer, and it is the pointer that is used to access the node.

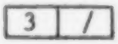
- The arrows in the diagram are used to represent pointers. Notice that the next address field of the last node on the list contains the special value called NIL.

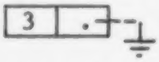
  - This NIL pointer is used to indicate the end of the list and it is not a valid address.

  - The entire linear linked list is accessed from the pointer "list" which points to (contains the address of) the first node in the list.

- The NIL pointer is sometimes pictured using a slash or the electronic symbol for ground. The following nodes are equivalent:







- We will usually use the first representation in our lessons.

3. Because the linear linked list is a dynamic data structure, we do not need to specify how many nodes are on the list.

   - The size of the list and the related amount of storage space used changes as we insert or delete nodes. In theory, new nodes can be inserted without limit; in practice, insertions are limited by available memory and the compiler's implementation.

   - With dynamic allocation we do not have to specify a size for our data structure as we did with the static allocation of our arrays.

   - Also, when we use dynamic allocation for our data structure, the computer allocates storage space to new nodes as they are created and recaptures storage space from nodes that are deleted.

4. A pointer can be used as part of a node (for example, the next address field) or by itself as in the case with the pointer called "list").

   - The pointer "list " is an example of a pointer variable.

     - Pointer variables take on values like other types of variables, but their values are addresses of nodes.

     - Pointers are very important because a node and the information contained within it can only be accessed by using a pointer. The reason is that nodes do not have names. Nodes are only pointed to by (that is, have their addresses contained within) pointers.

     - Since pointer variables have names (for example, "list"), we can use them to access the nodes they point to.

- A pointer variable can contain the value NIL. If the pointer variable which points to the first element of our list has the value NIL, then our list contains no nodes. Such a list is called an empty or NIL list. An empty list is pictured as:

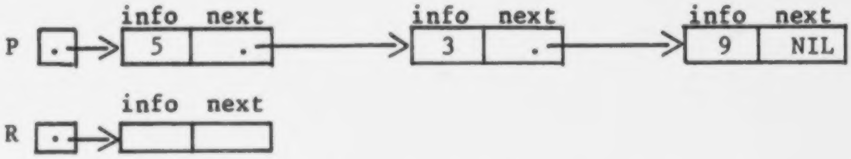
  list $\boxed{\text{NIL}}$

- If we have a pointer variable named "list" which points to our linear linked list, we can create an empty list with the following assignment:

```
list := NIL;
```

5. The algorithmic notations which follow are not valid Pascal procedures; in subsequent lessons we will learn how to modify them so they are. By using this notation, we can study and illustrate the concepts of linked lists without worrying about the details of Pascal syntax.

   - The algorithms which follow refer to the diagram below.



   - node (P) refers to the node pointed to by the pointer P. In the example above, node (P) refers to the node which contains the integer 5.

   - info (P) refers to the contents of the information field of the node pointed to by the pointer P. In the example, info (P) is the integer 5.

   - next (P) refers to the pointer which is contained in the next address field of the node pointed to by the pointer P. In the example, next (P) is the pointer which points to the node containing the integer 3.

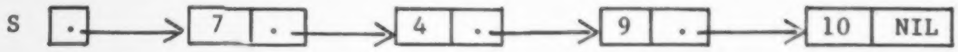
   - R := getnode is an operation which obtains an empty node and sets the contents of a variable pointer named R to the address of that node; that is, it has R point to the newly created empty node.

   - freenode (R) is an operation which makes the node pointed to by R available for reuse.

     - You can view getnode as an operation which creates new nodes and freenode as an operation which destroys nodes.

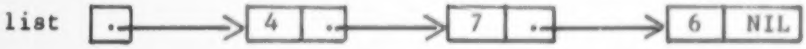
     - Once freenode (R) has been performed, the pointer R no longer points to any node. At this point it is illegal to reference node (R), info (R), next (R) or even R itself. The pointer R has no value, not even NIL. You may assign a value to R or create a new node at R using the getnode operation, but you may not reference (use the value of) R until it is reassigned a value.

6. Given the following linked list,



- info (S) is 7
- next (S) is the pointer which points to the node containing 4
- info (next (S)) is 4
- node (next (next (S)) is the node which contains 9
- Notice that all references to nodes, pointers and information must use a pointer variable (S in this case).

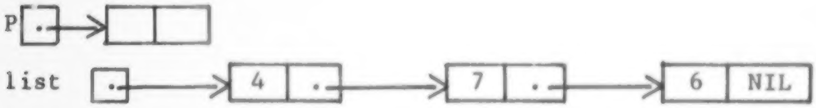
7. Given the linked list "list",



- Go through the steps necessary to add a new node containing the integer 9 to the front of the list. For each step, show the result and give an explanation. Use another pointer variable, called P, for this algorithm.
- Step 1: Obtain an empty node and set the pointer variable P to the address of the new node.

  P := getnode

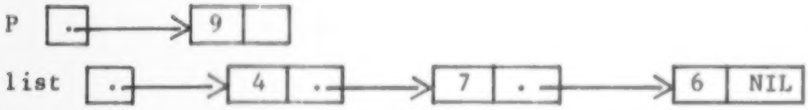
  Result:



- Step 2: Set the information portion of node (P) to 9.

  info (P) := 9

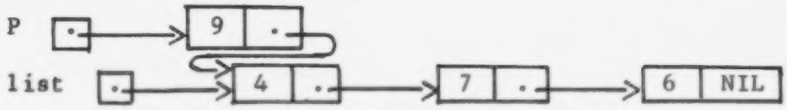
  Result:



- Step 3: Set the next address portion of node (P) to the original first node. That node will now be the new second node.

  next (P) := list

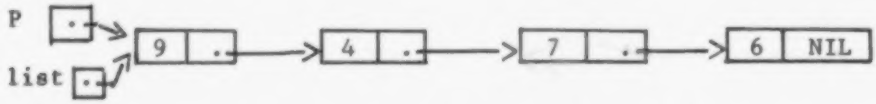
Result:



. Step 4: Set the pointer to the list (i.e. "list") to the new first node. At this point, the pointer P is no longer needed. It may be reassigned or reused to insert another node at the front of the list.

```
list := P
```

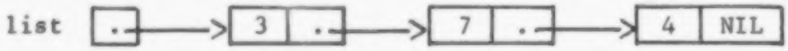
Result:



. Our complete algorithm to insert a node with the value 9 at the front of a linear linked list is:

```
P := getnode;
info (P) := 9;
next (P) := list;
list := P
```

8. Given the following linked list,



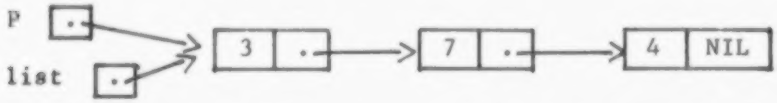
. Remove the first node after setting its value (information field) to the integer variable X. Again we will use a pointer variable called P.

. Step 1: First we need to assign the list to the pointer variable P. The need for P will be seen in the last step of the algorithm. The integer variable X is initially empty.

```
P := list
```

Result:

X =

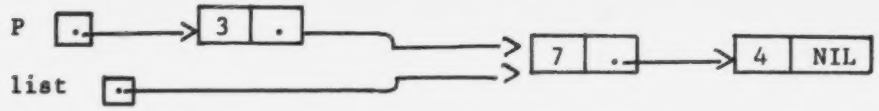


. Step 2: Set "list" to the front of what will become the new list.

```
list := next (P)     or     list := next (list)
```

Result:

X =

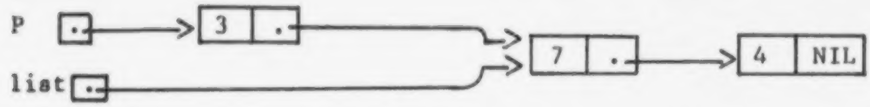


- Step 3: Assign the value in the node pointed to by P to the integer variable X.

```
X := info (P)
```

Result:

X = 3

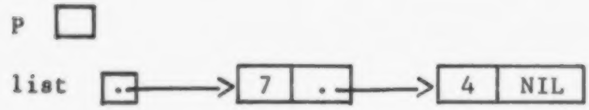


- Step 4: Remove the node pointed to by P from the list.

```
freenode (P)
```

Result:

X = 3



- Since freenode (P) destroys node (P) -- that is, returns it to the system to be reused -- the pointer variable P was needed. Without using the freenode operation, we could have done the following:

```
X := info (list);
list := next (list)
```

- But this makes the storage space for node (P) unavailable for future use.

9. Both of the algorithms above work only if the list is non-empty. The assignments to X and list are illegal if list is initially set to NIL. The following algorithm prevents such illegal assignments.

```
IF list = NIL THEN
  writeln ('list underflow')
ELSE
  BEGIN
    P := list;
    list := next (P);
    X := info (P);
    freenode (P)
  END;
```

10. Anything beyond these two previous algorithms becomes more complicated. But it is helpful to remember the following basic rules when you work with linear linked lists.

    - Check for an underflow condition whenever you delete a node.

    - Make sure your algorithm works for all lists, even those that are empty or contain just one node.

    - When you insert a node, you usually need a pointer pointing to the node immediately before this new node.

    - When you delete a node, you usually need a pointer pointing to the node immediately after the node to be deleted.

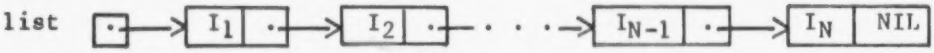
    - To get information from a node or to change a value in a node, you usually need a pointer pointing to that node.

    - When moving a pointer in a linear linked list, you can only move from the front towards the end of the list. Because you cannot move backwards, it's inadvisable to move the pointer too far.

    - If P is pointing to any but the last node, then P := next (P) will get P to point to the next node in the list.

11. Write a procedure to insert a node, which has a value of 9, at the end of a list called "list". Use the pointer variable T for creating the new node and P for determining where to insert it. Write the procedure so that it works with any generalized list named "list".

    - The generalized list which follows contains N elements. Remember -- the algorithm must work for all values of N, including when N is 0 (i.e. the empty list).



    - Since we want to insert the node at the end of our list, we must somehow get the pointer P to point to this last node (i.e., the node containing I_N). The only time that this is not possible is when N = 0. We must handle this case separately.

```
T := getnode;
info (T) := 9;
next (T) := NIL;
IF list = NIL THEN
  list := T
ELSE
  BEGIN
    P := list;
    WHILE next (P) <> NIL DO
      P := next (P);
    next (P) := T
  END;
```

- If the list is not empty we want P to point to the last node. When P is pointing to the last node, next (P) will be NIL. Therefore, we use the WHILE loop to move P along the list whenever next (P) is not equal to NIL. You will use algorithms like this whenever you work with linear linked lists.

Summary:

1. What are the two major disadvantages of using arrays to represent stacks and queues?

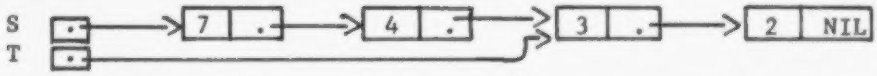
2. Define the following: linear linked list, node, pointer, information field, next address field and NIL?

3. What do the following terms refer to: node (P), info (P), next (P)?

4. What actions do the operations getnode and freenode perform?

Exercises:

1.



- Using the list S above, what do the following refer to?

    - info (S)
    - next (T)
    - next (next (T))
    - node (next (S))
    - info (next (T))

2. Write an algorithm that will free (i.e. destroy or return) all the nodes in list P.

3. Write an algorithm to delete the Nth node from a list Q. You may assume that N $>$ 0 and that the Nth node does exist.

4. Modify problem #3 so that the assumption is not necessary. That is, have the algorithm return an error message of 'N<= 0' or 'node does not exist' if these conditions occur.

5. Write an algorithm that determines the number of elements in list Q.

6. Write an algorithm that determines the sum of the integers in list Q.

7. Write an algorithm that determines the average of the integer elements in list Q.

TOPIC 71

STACKS AND QUEUES USING LINKED LISTS

Performance Objectives - Students will be able to:

- describe the linked list representation of stacks
- describe the linked list representation of queues
- discuss how an available list can be used to implement the getnode and freenode operations

Vocabulary: available list

Motivation:

- Write an algorithm to delete the last node from the linked list "list". Assume "list" has more than one node. Set X equal to the value of the last node. Get a pointer to point to the next to last node.

```
P := list;
WHILE next (next(P))<>NIL DO
  P := next(P);      (* P points to next to last node *)
Q := next (P);    (* Q points to last node *)
X := info (Q);
next (P) := NIL;
freenode (Q);
```

- The algorithm above deletes the last node from "list" as long as the linked list has more than one node. If "list" is empty or has just one node, the algorithm will not work because next (next(P)) will be undefined. The following algorithm works for all cases:

```
P := list;
IF P = NIL THEN
  writeln ('List underflow')
ELSE
  IF next (P) = NIL THEN
    BEGIN
      X := info (P);
      list := NIL;
      freenode (Q)
    END
  ELSE
    BEGIN
      WHILE next (next(P))<>NIL DO
        P := next(P);
      Q := next (P);
      X := info (Q);
      next (P) := NIL;
      freenode (Q)
    END;
```

Aim: How can linked lists be used to represent stacks and queues?

Development:

1. Consider a linear linked list where nodes are added and removed from the front of the list. It may be viewed as a stack where the first node on the list is the top of the stack.

   - We can push an element onto the stack by adding an element to the front of the linked list, and we can pop the stack by removing the first element from the linked list.

   - If S is a pointer to the first element of our linked list (viewed as a stack), then push (S, X) denotes the operation of pushing an element with value X onto stack S. We will use X := pop(S) to denote the operation of popping the stack and setting the variable X to the value of the element being removed. Finally, the operation empty(S) will test whether or not S equals NIL.

2. We can use algorithms similar to those in the last lesson to implement the push and pop operations. We can implement push (S, X) as follows:

```
P := getnode;
info(P) := X;
next(P) := S;
S := P;
```

   - We can implement X := pop(S) as follows:

```
IF empty (S) THEN
  writeln ('Stack underflow')
ELSE
  BEGIN
    P := S;
    S := next(P);
    X := info(P);
    freenode (P)
  END;
```

3. Consider a linear linked list where we add nodes to the end of the list and remove them from the front of the list. We may view this linked list as a queue where the first node on the list is the front of the queue and the last node is the rear of the queue.

   - We insert element X onto queue Q, denoted by insert (Q, X), by adding a node with the value X at the end of the linked list. We delete an element from the queue and set its value to the variable X, denoted by remove (Q, X), by removing the first node from the linked list.

   - In the motivational problem, we had to go through the entire list in order to get to the last node on the linked list. This can be very time consuming if our queue is long.

- To avoid this problem we usually have two pointers when we use a linked list to represent a queue. One points to the first node and the other to the last node. We can call them Q.front and Q.rear, respectively.
  - They both have the value NIL when the queue is empty. When writing algorithms for queues, it is important to check that both pointers are correctly set when you insert the first element onto an empty queue or delete the last element from a queue and leave it empty. The operation remove (Q, X) can be implemented by the following algorithm:

```
IF empty (Q) THEN
  writeln ('Queue underflow')
ELSE
  BEGIN
    P := Q.front;
    X := info(P);
    Q.front := next(P);
    IF Q.front = NIL THEN
      Q.rear := NIL;
    freenode (P)
  END;
```

- The operation insert (Q,X) can be implemented by the following algorithm:

```
P := getnode;
info(P) := X;
next(P) := NIL;
IF Q.rear = NIL THEN
  Q.front := P
ELSE
  next (Q.rear) := P;
Q.rear := P;
```

4. The getnode operation somehow creates a node and the freenode operation somehow destroys nodes that are no longer needed. This implies that we have an infinite supply of new nodes, which is obviously not true since a computer's memory is finite.

   - In reality, there is a finite collection of nodes that can be used. The getnode operation gets one of these available nodes if we have not used them all up already. The freenode operation returns an unneeded node back to this collection of available nodes.

   - This list of available nodes is called the <u>available list</u>. We usually view (and implement) this available list as a linked list being used as a stack. Therefore, freenode pushes a node onto the available list and getnode pops the available list.

   - Let us consider the available list just described having a pointer named "avail" pointing to the top of the stack. The operation freenode (P) can be implemented as follows:

```
      next(P) := avail;
      avail := P;
```

- The operation P := getnode can be implemented by:

```
    IF avail = NIL THEN
      writeln ('Overflow')
    ELSE
      BEGIN
        P := avail;
        avail := next (avail)
      END;
```

5. An overflow condition occurs if we use the getnode operation when the available list is empty. This is similar to the overflow condition that occurs in the array implementation of stacks and queues when we exceed the dimension of the array.

   - If we have just one stack or queue being implemented, the overflow condition occurs when we exceed the size of our array or available list depending on the implementation.

   - When several stacks or queues are being implemented there is an advantage to using an available list. We can use one available list to supply nodes to all of our stacks and queues. We will only get an overflow when the entire available list is depleted.

     - If we use arrays to implement the stacks and queues, we must specify the maximum space for each stack and queue individually. This division of the space available to us allows an overflow to occur when any one of the stacks or queues exceeds its individual maximum space, even though collectively they have not exceeded all the space that is available for their use. Thus, the available list allows our data structures to share a common storage allocation.

Summary:

1. How can we use a linked list to represent a stack?

2. How can we use a linked list to represent a queue?

3. What is an available list?

4. How can we implement getnode and freenode operations with an available list?

Exercises:

1. If we have a pointer C and a linear linked list pointed to by F, write an algorithm that creates a copy of the list F and has C point to this copy.

2. Given list L, write an algorithm that deletes every second element.

3. Write an algorithm that adds the elements of list L2 to the end of list L1.

TOPIC 72

PASCAL IMPLEMENTATION OF LINKED LISTS

Performance Objectives - Students will be able to:

- indicate the advantages and disadvantages of using lists in place of arrays
- show how a linked list can be implemented in Pascal using an array of records
- write Pascal subprograms that illustrate the freenode and getnode operations

Motivation:

- List two disadvantages and one advantage of using linked lists in place of arrays to implement stacks and queues.
  - One disadvantage is that linked lists require more storage space than arrays. With linked lists we need space not only for the information, but also for the pointers associated with each piece of information.
  - Another disadvantage is the additional time necessary to maintain the available list. Each addition and deletion for the linked list requires a corresponding deletion or addition for the available list.
  - An advantage of using linked lists instead of arrays is that one available list can be set up to supply nodes to all of our lists. By using one available list we specify the total space needed for all of our lists. On the other hand, array sizes must be specified individually and we could get an overflow for one array even though there is unused space left in the other arrays.

Aim: How can linked lists be implemented in Pascal?

Development:

1. Since each node has two parts (information and next address), we can represent nodes as records. Our entire collection of nodes will be represented as an array of these records.
   - Since we must specify the dimension of the array before the program executes and this in turn limits the number of nodes available, our implementation is again a static one.
   - Consider the address of a node to be its subscript in the array of nodes. Therefore, the next address fields and all external pointers are variables of the same type as the subscripts of the array. Use positive integers as subscripts and pointer values. Also use the integer 0 to represent the NIL pointer.
2. The following declarations may be used in Pascal to set up 100 nodes where the information portion of each node is an integer:

```
CONST
  numnodes = 100;
TYPE
  nodeptr = 0..numnodes;
  nodetype =
    RECORD
      info : integer;
      next : nodeptr
    END;
VAR
  node : ARRAY[1..numnodes] OF nodetype;
  list, avail : nodeptr;
```

3. While the nodes are numbered from 1 to 100, the value of the pointers (nodeptr) range from 0 to 100. This allows for the value of 0 for the NIL pointer.

   - The variables list and avail are pointer variables used to point to the linked list and available list, respectively. The above declarations are global so that both lists are accessible throughout the program.

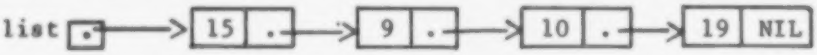
4. The Pascal expressions node[P], node[P].info and node[P].next are used to reference node (P), info (P) and next (P), respectively. By using these Pascal expressions and the freenode and getnode operations, we can easily convert the algorithms of our last two lessons into Pascal procedures or functions.

5. Consider the following array of records:

	info	next
1		
2		
3	9	8
4		
5	19	0
6		
7	15	3
8	10	5
9		

   - If list = 7, the above array of records can be considered to contain a linked list of four nodes.
     - Since list is 7, the node with subscript 7 (i.e. containing the value 15) is the first node of our list. This node also has a pointer (i.e. node[7].next) with a value of 3.
     - This indicates that the second node of our list is node[3] which has a value of 9.
     - The third node is node[8] which has a value of 10 and the fourth node is node[5] which has the value 19.

- Since node[5].next is 0, node[5] is the last node of our linked list. Therefore, our linked list "list" is equivalent to:



6. Several lists can be contained in the same array of records. Consider the following array and pointers:

	info	next
1	6	8
2		
3	7	10
4	10	0
5		
6	1	16
7		
8	7	0
9		
10	12	17
11		
12	11	0
13		
14	9	0
15		
16	2	20
17	3	4
18		
19		
20	5	12

list1 = 3
list2 = 1
list3 = 14
list4 = 6

. The linked list list1 starts at node[3] and contains the integers 7, 12, 3 and 10. The list list2 starts at node [1] and contains the integers 6 and 7. The list list3 has only the node node[14] with the value 9. Finally, list list4 starts at node[6] and contains the integers 1, 2, 5 and 11.

. As a result, our single array is being used to represent four different linked lists. More lists can be represented in the above array as long as there are unused nodes available.

7. If we want to add a node to one of the lists in the previous example we could use any of the unused nodes in the array. We usually keep the unused nodes in a linked list (the available list) pointed to by a pointer variable. We will use the pointer variable avail for our available list.

. At the start of our program all nodes are unused and all lists are empty. We set the lists as initially empty by setting their pointers to 0 or NIL (e.g. list := 0). We can set up available list avail to consist of all the nodes in our array by the following initialization procedure:

```
avail := 1;
FOR I := 1 TO numnodes -1 DO
  node[I].next := I + 1;
node[numnodes].next := 0;
```

- Our one array will therefore contain all of our lists, including the available list. In the above initialization routine for avail we have linked the nodes in sequential order. This initial ordering was established solely for convenience, therefore any other ordering that links all the freenodes to one another will work equally as well.

8. Now that we have created and initialized our available list we can implement our freenode and getnode operations in Pascal. We will set up a procedure freenode to accept a pointer to a node and return that node to the available list.

```
PROCEDURE Freenode (P : nodeptr);
BEGIN
  node[P].next := avail;
  avail := P
END;
```

- Implement getnode as a function that removes a node from the available list and returns a pointer to it. By implementing getnode as a function, rather than a procedure, our previous algorithmic notation does not have to be changed (e.g. P := getnode).

```
FUNCTION Getnode : nodeptr;
BEGIN
  IF avail = 0 THEN
    writeln ('List overflow')
  ELSE
    BEGIN
      getnode := avail;
      avail := node[avail].next
    END
END;
```

9. The declaration set up in this lesson enables us to implement linked lists in Pascal. By modifying the algorithms in the previous lesson, similar to the way we just modified getnode and freenode, we can use our linked lists to implement stacks and queues.

- When you write a Pascal program using linked lists, you should globally declare the array of records and all pointer variables that point to the first node of each list. Your program should include a getnode function and a freenode procedure. You may also want to include the boolean function empty. Your program should also include procedures that will insert and delete nodes from your list.
  - The manner of insertion and deletion will depend on whether you are using the linked list or array to implement a stack or queue.

- At the beginning of your program or in an initialization procedure be sure to initialize your available list and set all other list pointers to NIL.

Summary:

1. What are the two major disadvantages of using linked lists in place of arrays to implement stacks and queues?

2. What is the major advantage of using linked lists in place of arrays to implement stacks and queues?

3. How can we implement a node in Pascal?

4. How can we implement a linked list in Pascal?

Exercises:

1. Write a boolean function to implement the empty (P) function. It should return the value true when list P is empty and false otherwise. Use the notations from this lesson.

2. Using the declarations from this lesson, write a procedure to implement push (S, X) from the last lesson.

3. Using the declarations from this lesson and the empty function from exercise 1, write a function to implement pop(S) from the last lesson.

4. Write a procedure, called printlist (P), which will print out all the elements of a list P starting at the first element and going through the list sequentially. If list P is empty, print out "Empty list".

5. Write a program that implements stack S as a linked list using an array of records. Your program should input integer X and perform the following actions depending on the value of X.

   - If X is positive, then push (S, X), print the entire stack from the top and then input another value for X and continue the process.

   - If X is negative, then pop(S), print the value of the node popped, print the entire stack from the top and then input another value for X and continue the process.

   - If X is 0, then print out the list and terminate the program.

   - Write this program using the declarations from this lesson, the freenode procedure, the getnode function and your solutions to the first four exercises.

6. Using the declarations from this lesson, write a procedure to implement insert (Q, X) from the last lesson.

7. Using the declarations from this lesson, write a Pascal function to implement remove (Q, X) from the last lesson.

8. Write a Pascal program that implements queue Q as a linked list using an array of records. The program should operate in the same manner as that of exercise 5, except push (S, X) and pop(S) should be replaced by insert (Q, X) and remove(Q, X), respectively. Also, the list should now operate in the FIFO fashion of a queue, instead of the LIFO fashion of a stack.

TOPIC 73

LINKED LIST OPERATIONS IN PASCAL

Performance Objectives - Students will be able to:

- discuss problems related to the insertion and deletion of items in a linked list
- write Pascal procedures that insert and delete items in a linked list
- describe how an item can be placed in its proper position in a sorted linked list

Motivation:

- Using the declarations and operations from the last lesson, write a Pascal procedure to insert a node, with the value X, at the end of list Q. Call this procedure insertatend (Q, X). Model the procedure after the algorithm developed in a prior lesson.

- This is one possible solution to the problem:

```
PROCEDURE Insertatend (VAR Q : nodeptr, X : integer);
VAR
  P, T : nodeptr;
BEGIN
  T := getnode;
  node[T].info := X;
  node[T].next := 0;
  IF Q = 0 THEN
    Q := T
  ELSE
    BEGIN
      P := Q;
      WHILE node[P].next<>0 DO
        P := node[P].next;
      node[P].next := T
    END
END;
```

Aim: How can some frequently used list operations be coded in Pascal?

Development:

1. We have previously used linked lists to implement stacks and queues, but linked lists are also important as a data structure in their own right.

   - A linked list is basically a data structure in which each element points to the element following it. We get to elements in a linked list by starting at the first element of the list and following the pointer from each element to the next element.

- Whereas array elements can be obtained by random access, linked list elements can only be accessed sequentially. This type of access is the major disadvantage of using linked lists.

- The major advantage of using linked lists is that elements can be easily inserted and deleted. When we insert an element into the middle of an array we have to move all elements that follow the inserted one down. Similarly, whenever we remove an element from an array we have to move all the elements which follow the deleted element up. These operations take up a great deal of computer time if the array has many elements.

2. Insertions and deletions can be performed in linked lists without moving any of the other elements. We only have to change one of the previously established pointers to perform an insertion or deletion anywhere in the list. The following algorithm can be used to insert a node with a value of X after the node pointed to by P:

```
Q := getnode;
node[Q].info := X;
node[Q].next := node[P].next;
node[P].next := Q;
```

- The above algorithm can be used to insert a node anywhere in our list. Similarly, we can delete the node following node[P] and assign its contents to the variable X with the following algorithm:

```
Q := node[P].next;
X := node[Q].info;
node[P].next := node[Q].next;
freenode(Q);
```

3. The major problem with insertions and deletions in linked lists is that of getting a pointer (like P above) to point to the correct node.

- When you insert a node into a linked list you must get a pointer to point to the node immediately preceding the new node. This has to be done because we must change the pointer of the preceding node to point to our new node.

- When you delete a node from a linked list you must also get a pointer to point to the node immediately preceding the node to be deleted. Again, this has to be done because we must change the pointer of the preceding node.

- We must always get a pointer to point to the earliest node on the list that is to be modified since we can only go forward on a list.

4. In the motivational problem we were inserting a node at the end of a linked list. For the reason given above, we had to get a pointer to point to the last node on the list.

- The WHILE loop moved P until it pointed to the last node on the list. Likewise, all insertions and deletions require moving a pointer so it points to the node preceding the one being inserted or deleted.

5. We will continue to use the declarations and operations from the last lesson in our next few Pascal procedures.

   - Our first procedure deletes all nodes with the value of X from the list R. Since we must know the predecessor of a node to delete it, we need two pointers, P and Q. We traverse the list with P, while Q always points to the predecessor of P. This technique of using two pointers, one following the other, is very common and useful when working with lists. The following procedure performs the required deletions:

```
PROCEDURE Deletex (VAR R : nodeptr, X : integer);
VAR
  P, Q : nodeptr;
BEGIN
  Q := 0;
  P := R;
  WHILE P<>0 DO
    IF node[P].info = X THEN
      IF Q = 0 THEN   (* delete first node of list *)
        BEGIN
          R := node[P].next;
          freenode (P);
          P := R
        END
      ELSE (* remove node following Q *)
        BEGIN
          node[Q].next := node[P].next;
          freenode (P);
          P := node[Q].next
        END
    ELSE (* continue traversing list *)
      BEGIN
        Q := P;
        P := node[P].next
      END
END;
```

6. Notice that we had to handle two separate types of deletions.

   - One group of statements deletes the first node from the list while another group deletes a node that is not the first node. This has to be done because in the first case the list pointer R has to be changed, and in the second case a next address field pointer has to be changed.

   - All linked list operations must take these special cases into consideration, and most operations will have to handle these cases with separate groups of statements. This is another problem that we face when we insert or delete elements from linked lists.

7. It is often useful to maintain a sorted (ordered) linked list; that is, a linked list where each element is smaller than or equal to all the elements which follow it on the list.

   - Searching for an element on a sorted linked list requires less time if the element is not on the list because we can stop looking once we reach a value that is larger than the one being searched for.

   - In an unsorted linked list we must search the entire list before we can determine that the value is not on the list.

   - To insert an element in its proper place in a sorted linked list we must move a pointer along the list until we find a value larger than the one being inserted. We must insert our node before the one with the larger value. Since we need a pointer to the node preceding the one with the larger value, we could again use our two-pointer technique.

   - Special considerations should be made for the cases where the node to be inserted will be the first or last node on the list. This insertion procedure will be covered in one of the exercises.

8. Until now, all of our linked lists have had integer values. It should be obvious, by the way we declared nodetype in the last lesson, that our nodes could easily be changed to contain values other than integer. For example:

```
nodetype =
  RECORD
    info : PACKED ARRAY[1..20] OF char;
    next : nodeptr
  END;
```

   - Each node could even have several fields of information as the following declaration illustrates.

```
nodetype =
  RECORD
    info :
            RECORD
              name : PACKED ARRAY[1..30] of char;
              age : integer;
              sex : char
            END;
    next : nodeptr
  END;
```

Summary:

1. What are the two major problems related to the insertion and deletion of items in a linked list?

2. What is the major advantage of performing an insertion (or deletion) in a linked list rather than an array?

Exercises:

1. Using the declarations and operations from the last lesson, write a Pascal procedure to delete the last node from the linked list Q and assign its contents to the variable X.

2. Using the declarations and operations from the last lesson, write a Pascal procedure to insert a node with a value of X in its proper place in the sorted linked list L.

TOPIC 74

CIRCULAR LINKED LISTS

Performance Objectives - Students will be able to:

- define a circular linked list
- discuss operations on circular linked lists
- state the Josephus Problem
- describe an algorithm that uses a circular linked list to solve the Josephus problem

Vocabulary: circular linked list, Josephus Problem

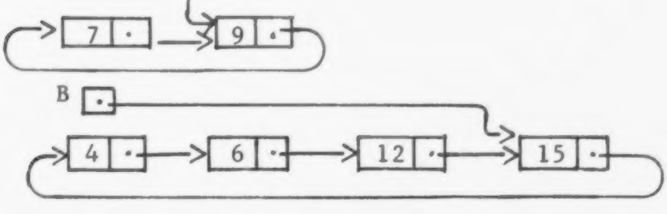
Motivation:

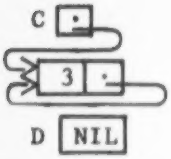
- How can we modify our linear linked list data structure to represent a circular arrangement of elements (like people sitting around a circular table)?
- One solution is to have the pointer from the last element of the list point back to the first element of the list.
- The linked list described above, where the last element points to the first element, is called a circular list or circular linked list.

Aim: How can a circular linked list be used to solve problems?

Development:

1. A circular list does not have a natural "first" or "last" node.
   - Whenever we want to establish a first and last node, we usually have a pointer set to the last node. When the pointer is set to the last node of a circular list we can easily get to both the last and first node of the list.
   - If, on the other hand, we set the pointer to the first node we have to traverse the entire list to get to the last node. The pointer to the last node is set to NIL when the circular linked list is empty.
2. The following diagrams illustrate the circular lists A, B, C and D with 2, 4, 1 and 0 elements, respectively.





3. We can insert and delete elements in a circular list with procedures similar to those for linear linked lists. There are slight differences in the procedures and certain modifications must be made because of the pointer from the last node to the first node.

   . The major difference in traversing a circular list is determining when we have completely traversed the list. For example, assume we have a circular list with the pointer named first pointing to the first node. If P is a pointer traversing the list, P points to the last node of the list when next (P) = first. Similarly, the circular list named list consists of just one element when next (list) = list.

4. Using the conventions established for circular lists along with the declarations and operations from the last two lessons for linear linked lists, write a procedure to delete the first node from the circular list list (where list points to the last node) and set the variable X equal to the value of the deleted node.

```
PROCEDURE Deletefirst (VAR last : nodeptr; VAR X : integer);
VAR
  Q : nodeptr;
BEGIN
  IF last = 0 THEN
    writeln ('List Underflow')
  ELSE
    IF last = node[last].next THEN
      BEGIN
        X := node[last].info;
        freenode (last);
        last := 0
      END
    ELSE
      BEGIN
        Q := node[last].next;
        X := node[Q].info;
        node[last].next := node[Q].next;
        freenode (Q)
      END
END;
```

5. Now consider a problem known as the <u>Josephus Problem</u>. The problem states that we have a group of soldiers surrounded by an overwhelming enemy force. There is no hope for victory without reinforcements, but there is only a single horse available for escape. The soldiers agree to a pact to determine which of them is to escape and summon help.

- They form a circle and a number N is picked from a hat. One of their names is also picked from a hat. Beginning with the soldier whose name is picked, they begin to count clockwise around the circle. When the count reaches N, that soldier is removed from the circle, and the count begins with the next man. The process continues so that each time the count reaches N, a man is removed from the circle. Once a soldier is removed from the circle, of course, he is no longer counted. The last soldier remaining is to take the horse and escape.

- The problem is -- given a number N, the ordering of the men in the circle, and the man from whom the count begins -- to determine the order in which men are eliminated from the circle and which man escapes.

6. The Josephus Problem can be solved in a straight forward manner by using a circular list. Each node will contain the name of one man and a pointer to the man next to him in the clockwise position.

   - Input the number N and a list of names that is the clockwise ordering of the men in the circle, beginning with the man from whom the count is to start. The last input line contains the string 'end', indicating the end of the input. Print the names of the men in the order that they are eliminated and the name of the man who escapes. The following program outline could be used to solve the problem:

```
read (N);
read (name);
WHILE name<>'end' DO
  BEGIN
    insert name on the circular list;
    read (name)
  END;
WHILE there is more than one node on the list DO
  BEGIN
    count through N-1 nodes on the list;
    print the name in the Nth node;
    delete the Nth node
  END;
print the name of the only node on the list
```

7. Circular lists can also be used to represent stacks and queues. For stacks, we push by adding an element to the front of the list and pop by deleting the first element. For queues, we insert elements to the rear of the list and delete elements from the front of the list.

Summary:

1. What is a circular linked list?

2. What is the Josephus Problem?

3. How can we solve the Josephus Problem?

Exercises:

For the following exercises use the conventions established for circular lists along with the declarations and operations from the last two lessons:

1. Write a procedure that prints out the values of circular list P, starting at the first element and ending at the last element.

2. Write a procedure that inserts a node with the value X at the front of circular list P.

3. Write a procedure that inserts a node with the value X at the end of circular list P.

4. Write a Pascal program that implements stack S as a circular list. Your program should input integer X and perform one of the following actions depending on the value of X.

   - If X is positive, then push (S, X), print the entire stack from the top and then input another value of X and continue the process.

   - If X is negative, then pop(S), print the value of the node popped, print the entire stack and then input another value for X and continue the process.

   - If X is 0, then print out the stack and terminate the program.

5. Write a Pascal program that implements queue Q as a circular list. The program should operate in the same manner as that of exercise 4, except push (S, X) and pop(S) should be replaced by insert (Q,X) and remove (Q,X), respectively.

6. Write a Pascal program that will solve the Josephus Problem. Follow the outline given in this lesson.

TOPIC 75

HEADER NODES AND DOUBLY LINKED LISTS

Performace Objectives - Students will be able to:

- define a header node
- list several applications of header nodes
- discuss how a doubly linked list eliminates several drawbacks of a linear linked list.

Vocabulary: header node, doubly linked list

Motivation:

- Write an algorithm to delete the node pointed to by P, if such a node exists, from a circular list. Let P then point to the preceding node.

- One possible solution is:

```
IF P = NIL THEN
  writeln ('List Underflow')
ELSE
  IF P = next (P) THEN
    BEGIN
      freenode (P);
      P := NIL
    END
  ELSE
    BEGIN
      Q := P;
      WHILE next (Q)<>P DO
        Q := next (Q);
      next (Q) := next (P);
      freenode (P);
      P := Q
    END;
```

Aim: How can a linked list be modified for particular applications?

Development:

1. The above algorithm points out two of the difficulties encountered with circular lists.

   - To delete node (P) we need a pointer to point to the node immediately preceding node (P). Since we can only move forward in a circular list, we must go around the entire circular list to get to the node immediately preceding node (P).

   - We also must maintain a stationary pointer (P in the above problem) to determine our position in the list. We can only determine that we have traversed the entire list by comparing next (Q) to P.

- These two problems can be overcome by modifiying our circular list. The modifications can be used with both linear and circular lists.

2. One way to modify the list is to keep an extra node, called a header node, in the list.

   - In a linear list, the header node is kept at the front of the list. In a circular list, it is kept between the last and first nodes. The external pointer to a circular list usually points to the header node, but other arrangements are possible.

   - Header nodes provide many benefits, among them the ability to supply additional information about the list. The header node is always present in a list, but it is not considered an element of the list. An empty list is, therefore, one which contains only the header node.

   - Some list operations, such as insertion and deletion, can be simplified when we use header nodes. This is because all lists, including the empty list, contain at least one node and many of the special provisions in our operations to handle the empty list can be simplified or eliminated completely.

   - The information field of the header node is usually used to store information about the list, but some applications do not use this field.

     - The type of information stored depends on the field type. If the field is of type integer, it could keep count of the number of nodes in the list. If this is the case, insertion and deletion operations must be modified to update the count.

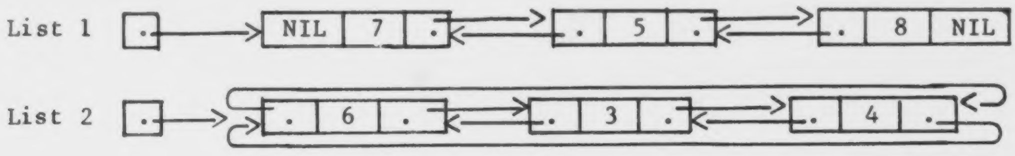
   - Usually, in a circular list, the header node's information field is set to a special value, one which is not valid for a list node. When this is done, we do not need a pointer set to the header while traversing a list because the header can be distinguished from a regular list node.

3. Another way to modify a list is to have each node contain two pointers.

   - One is used to point to the next node in the list while the other points to the preceding node. With these two pointers we can traverse the list both forwards and backwards.

   - Such a list is called a doubly linked list. Doubly linked lists may be either linear or circular and may or may not contain a header node.

4. The following diagrams illustrate a linear doubly linked list and a circular doubly linked list, respectively.



- Nodes for doubly linked lists can be set up with declarations similar to the following:

```
node =
  RECORD
    info : integer;
    left,
    right : nodeptr
  END;
```

- In the above declaration, left and right are the pointers to the nodes on either side of the given node.

5. Insertions and deletions for doubly linked lists are usually more complicated than those for singly linked lists.

   - Whenever elements are inserted or deleted, two pointers must be altered rather than just one.

   - However, certain operations are easier because of the ability to easily get to the node immediately preceding a given node.

6. Compare the following algorithm with the solution to the motivational problem and you will see that the use of a doubly linked list has both advantages and disadvantages. Left (P) and right (P) represent the two pointers of node (P).

```
IF P = NIL THEN
  writeln ('List Underflow')
ELSE
  IF P = next (P) THEN
    BEGIN
      freenode (P);
      P := NIL
    END
  ELSE
    BEGIN
      Q := left (P);
      right (Q) := right (P);
      left (right (P)) := Q;
      freenode (P);
      P := Q
    END
```

<u>Summary</u>:

1. What is a header node?

2. What are some uses of header nodes?

3. What are some advantages of using header nodes?

4. What are some disadvantages of using header nodes?

5. What is a doubly linked list?

6. What is the major advantage of using a doubly linked list?

Exercises:

1. Write an algorithm to delete the first list node, if it exists, from linear list P which has a header node. Set the variable X to the value of the deleted node.

2. Write an algorithm to insert a node with value X after the header node of the circular list P.

3. Write an algorithm to insert a node with value X at the front of linear doubly linked list P (without header node).

4. Write an algorithm to insert a node with value X at the front of circular doubly linked list P (with header node).

TOPIC 76

POINTER VARIABLES

Performance Objectives - Students will be able to:

- cite the disadvantages of using an array implementation for linked lists
- show how pointer variables can be declared in Pascal
- describe how the NEW and DISPOSE procedures can be used to dynamically allocate and free storage
- discuss the problem of using a dangling pointer

Vocabulary: dynamic data structure, pointer variables, new, dispose, dangling pointer, NIL, (UCSD: mark, release)

Motivation:

- With an array implementation we must decide on the maximum number of nodes to allow when our program is written. What are the two major disadvantages of this fixed allocation?
  - First, no matter how many nodes we set up, it is still possible we will need more. Thus, we can "overflow" our linked list.
  - Second, all nodes declared must remain allocated to the program throughout its execution. That is, if we set up our array to have 1000 nodes (to avoid possible overflow) then the storage space for these 1000 nodes is reserved throughout the program, even when our linked list is empty or only has a few nodes.

Aim: What are pointer variables?

Development:

1. A data structure that is set up at the beginning of a program and remains the same throughout the program is static.
   - An array is an example of a static data structure because its size and basic structure cannot be changed, even though the values it contains can.
   - On the other hand, a dynamic data structure is a data structure that changes as the program executes. If necessary, the structure gets more storage space to grow or frees storage space that it no longer needs.
2. Certain features in Pascal allow us to dynamically create a linked list. With such a structure, we can create new nodes for our linked list as they are needed and release the storage space for nodes that are no longer being used.

3. Dynamic structures are possible because of pointer variables. A *pointer variable* is a variable whose value is an address. This address indicates what the pointer variable is pointing to.

   - These pointers are used much in the same way that we used subscripts for pointers in array implementations of stacks, queues and linked lists.

   - A pointer variable can be declared to contain the address of a certain type of data. For example, the declaration can state that it contain the address of an integer or a real.

   - Pointer variables which are declared to point to different types of data are themselves considered to be of different types. That is, a pointer to an integer is not the same type as a pointer to a real.

4. The following declarations show how pointer variables are declared:

```
VAR
P, Q : ↑integer;
R, S : ↑real;
```

NOTE: When running many versions of Pascal, such as Apple Pascal, the symbol ∧ is used in place of ↑ .

5. In the example above, the variables P and Q are integer pointers while R and S are real pointers. Note that P and Q are variables of the same type, whereas P and R are of a different type.

   - All four of these variables will be assigned addresses as their values. Therefore, R is a variable which contains the address of a real number; that is, R does not contain a real number itself, but only the address of the real number.

6. With the above declarations we have declared four pointer variables for addresses. At this time, no space has yet been reserved for the two integer and two real values. The space for these values will be created dynamically during the program execution.

   - The procedure that creates the space for data is called *new*. For example, the statement

     ```
     new (P);
     ```

     creates storage space for an integer (since P is an integer pointer) and puts the address of the storage space into the pointer variable P. Likewise, the statement

     ```
     new (S)
     ```

     creates storage space for a real and puts the address of the storage space into the pointer variable S.

- The storage space for the integer or real value created above can only be referenced by a pointer variable that contains its addresses. After the two 'new' statements above have been executed, the integer and real locations created are called P↑and S↑, respectively.

- To create the two data elements and give them the values 3 and 4.7, respectively, we must execute the following statements:

```
new (P);
new (S);
P↑ := 3;
S↑ := 4.7;
```

7. When we no longer need a data element we can free the storage space that it is using. This allows its storage space to be used for other data. The procedure that does this is called <u>dispose</u>. The statement:

```
dispose (P)
```

frees the space that was allocated to P↑. After dispose (P) has been called it is illegal to reference P↑.

- At this point P is called a <u>dangling pointer</u> because it no longer points to a valid element. Most implementations of Pascal do not give an error message when a dangling pointer is referenced. It is, therefore, up to the programmer to avoid the illegal referencing of a dangling pointer.

8. Pascal has the reserved word NIL for the value of a NIL pointer. We can set the pointer Q to NIL with the following statement:

```
Q := NIL;
```

9. We can have several pointers with the same address. When this occurs you should proceed with care. For example, the following statements may produce unexpected results:

```
new (P);
P ↑:= 7;
Q := P;
Q ↑:= 8;
writeln (P↑, '-', Q↑)
```

- The output from the above statements is 8-8 (not 7-8).

  - The first statement creates an integer data element and put its address in pointer variable P.

  - The second statement sets the integer data element just created to 7.

  - The third statement sets the address of Q to the address of P.

  - The fourth statement changes the integer data whose address is in Q (and in this case also in P) to the value 8.

- At this point the data element with a value of 8 has its address both in P and Q. Therefore, the last statement will output 8-8.
- In UCSD Pascal, the procedure new is implemented but not the procedure dispose. Instead, two procedures called mark and release are used for returning dynamic memory allocations to the system. We will not go into detail on the use of mark and release.

Summary:

1. What are the two major disadvantages of using an array implementation for a linked list?
2. What is the difference between static and dynamic storage allocation?
3. How do we declare pointer variables in Pascal?
4. What does new (P) do?
5. What does dispose (P) do?
6. What is a dangling pointer?

Exercises:

1. Tell what each of the following statements does if B and C are integer pointers, D and E are real pointers and F is an integer variable:
    - new (C)
    - B := C
    - dispose (B)
    - new (D)
    - D↑ := E↑
    - E↑ := F
    - E := NIL
    - E↑:= 3.4
    - F := B↑
2. Using B, C, D, E and F from exercise 1, tell whether or not the following are legal.
    - B := C↑
    - F := NIL
    - dispose (F)
    - 7 := E↑
    - E↑:= B↑
    - B↑:= NIL
    - dispose (D↑)
    - ↑E := 4.3
    - D := NIL
    - new (C↑)
    - E↑:= 5
    - E := B

TOPIC 77

DYNAMIC LINKED LIST

Performance Objectives - Students will be able to:

. explain how pointer variables can be used to define nodes of a linked list
. write a procedure to dynamically create a linked list
. write a procedure to insert elements in a linked list

Motivation:

. In our array implementation of linked lists, nodes were represented as records consisting of two fields. Describe the two fields.

- The first field, info, contained the information portion of the node. It was usually declared as type integer, but other types can be used.

- The second field, nodeptr, contained a pointer to the next node on the list. This pointer was an integer equal to the array subscript of the next node. Thus, integers acted as pointers in our array implementation.

Aim: How can dynamic variables be used to implement linked lists?

Development:

1. To create a dynamic linked list the nodes must be created when needed. The nodes will be very similar to those in the array implementation. The only difference is that the pointers used will be the pointer variables from the last lesson rather than integers.

2. The following declarations can be used to create a linked list:

```
TYPE
  nodeptr  = ↑node;
  node =
    RECORD
      info : integer;
      next : nodeptr
    END;
VAR
  list, temp : nodeptr;
```

3. A few points should be noted about the above declarations.

   . First notice that we have declared nodeptr to be a pointer to a node before we have even declared what a node is. This type of forward referencing is legal in Pascal.

     - Since nodeptr is an address, the compiler can assign storage space for nodeptr to keep an address. The type of node does not matter because all addresses require the same amount of storage space.

- Next notice that the only storage space allocated has been for the pointers list and temp. The storage space for nodes will be allocated dynamically as they are needed.

4. The procedures new and dispose will be used to create new nodes and to free storage space, respectively. These procedures were described in the last lesson, but we will review them again.

   - The procedure new (temp) allocates storage for a node and puts the address of the node into the variable temp.

   - The procedure dispose (temp) frees the storage space of the node whose address is in temp and leaves temp as a dangling pointer.

   - The new and dispose procedures eliminate the need for the getnode and freenode operations that we used with arrays.

     - The getnode and freenode operations could be implemented if needed, but their implementation is left as an exercise.

   - Also, since the computer maintains a list of the available storage space and allocates storage space for new nodes, the programmer does not have to be concerned with maintaining an available list.

5. To create an empty list, the value of NIL must first be assigned to the pointer variable which points to the first node of the list.

   - Use the pointer variable named list to point to the head of the list.

   - Initialize the linked list to the empty list with the following assignment statement: **list := NIL;**

   - It is important to initialize each list at the beginning of the program. If you do not, you will run into trouble when that list is first accessed.

6. Now create a node and insert it into the empty list. Write a procedure that inserts a new node with the value X at the head of the list "list".

```
PROCEDURE Insert (VAR list : nodeptr; X : integer);
VAR
  temp : nodeptr;
BEGIN
  new (temp);
  temp↑.info := X;
  temp↑.next := list;
  list := temp
END;
```

   - The procedure insert uses a local pointer variable named temp to point to the new node until it is inserted into the list.

- Notice that the parameter list is declared as a variable parameter. This must be done because the procedure insert has to change the global pointer variable which points to the head of the list.
  - If list were set up as a value parameter, our list would remain empty after each insertion because the reassignment to list would only change a local variable named list and not the global variable passed to the procedure.

Summary:

1. How are nodes declared for dynamic linked lists?

2. Do we have to maintain an available list for dynamic linked lists? Why?

3. How are dynamic linked lists initialized?

4. How do we insert nodes at the head of a dynamic linked list?

Exercises:

1. Write a boolean function to implement empty (list) for dynamic lists. The function should be true when the list named list is empty, and false otherwise.

2. Implement the function getnode for dynamic nodes. The function should create a new node and take on the value of the address of the newly created node.

3. Implement the procedure freenode (P) for dynamic nodes. The procedure should free the storage space allocated to the node whose address is in P.

TOPIC 78

LINKED LIST OPERATIONS

Performance Objectives - Students will be able to:

- write a procedure to delete an element from a dynamic linked list.
- write a procedure to insert and delete items at the end of a dynamic linked list.
- discuss the advantages and disadvantages of the dynamic and array implementations of a linked list.

Motivation:

- Using the declarations from the last lesson, write a procedure to insert a node at the end of a dynamically-implemented linked list. Your procedure should insert a new node with value X at the end of "list".

  One possible solution is:

```
PROCEDURE Tailinsert (VAR list : nodeptr; X : integer);
VAR
  temp, tail : nodeptr;
BEGIN
  new (temp);
  temp↑.info := X;
  temp↑.next := NIL;
  IF list = NIL THEN
    list := temp
  ELSE
    BEGIN
      tail := list;
      WHILE tail↑.next <> NIL DO
        tail := tail↑.next;
      tail↑.next := temp
    END
END;
```

Aim: How can routine operations be executed on dynamic linked lists?

Development:

1. Notice that the above procedure is very similar to the procedure used to insert a node at the end of an array-implemented linked list.
   - In fact, all of our procedures for array-implemented linked lists can be easily modified to handle dynamically-implemented linked lists.
2. Write a procedure to delete a node from the head of a linked list.

```
PROCEDURE Headdelete (VAR list : nodeptr);
VAR
  temp : nodeptr;
BEGIN
  IF list = NIL THEN
    writeln ('Underflow')
  ELSE
    BEGIN
      temp := list;
      list := list↑.next;
      dispose (temp)
    END
END;
```

3. When we delete a node from the tail end of a linked list, we must consider three possible cases.

   . The following procedure deletes a node from the end of a dynamically implemented linked list.

```
PROCEDURE Taildelete (VAR list : nodeptr);
VAR
  temp, tail : nodeptr;
BEGIN
  IF list = NIL THEN
    writeln ('Underflow')
  ELSE IF list↑.next = NIL THEN
    BEGIN
      dispose (list);
      list := NIL
    END
  ELSE
    BEGIN
      temp := list;
      (* Get temp to point to next to last node *)
      WHILE temp↑.next↑.next<> NIL DO
        temp := temp↑.next;
      tail := temp↑.next;
      temp↑.next := NIL;
      dispose (tail)
    END
END;
```

4. The procedures get a little more involved when we operate on nodes that are not at the beginning or end of our list.

   . For example, write a procedure to delete the first node with a given value from a list, if that value appears on the list.

```
PROCEDURE Delete (VAR list : nodeptr; X : integer);
VAR
  found : boolean;
  temp, notneeded : nodeptr;
```

```
BEGIN
  IF list = NIL THEN
    writeln ('Empty list - deletion not possible.')
  ELSE    (* first node contains X *)
    IF list↑.info = X THEN
      BEGIN
        notneeded := list;
        list := list↑.next;
        dispose (notneeded);
        writeln ('Node with value ', X, ' deleted.')
      END
    ELSE
      IF list↑.next = NIL THEN
      writeln (X, ' not on list - deletion not possible.')
      ELSE
        BEGIN
          temp := list;
          notneeded := list↑.next;
          IF notneeded↑.info = X THEN
            found := TRUE
          ELSE
            found := FALSE;
          WHILE (notneeded↑.next<>NIL) AND (NOT found) DO
            BEGIN
              temp := notneeded;
              notneeded := notneeded↑.next;
              IF notneeded↑.info = X THEN
                found := TRUE
            END;
          IF NOT found THEN
            writeln (X, ' not on list - deletion not possible.')
          ELSE
            BEGIN
              temp↑.next := notneeded↑.next;
              dispose (notneeded);
              writeln ('Node with value ', X, ' deleted.')
            END (* ELSE *)
        END (* ELSE *)
END; (* Delete *)
```

5. In the above procedure we handled three special cases.

   - The first case is that of the empty list. This case is handled separately in almost all procedures involving linked lists.

   - The second special case is that of the node being at the front of the list. This has to be handled separately because it is the only case where the external pointer named list must be changed.

- The third special case is that of the list containing only one node (and by the second special case we know that it is not the node we are trying to delete).
  - We handle this case separately because we plan to use two pointers to traverse the list. These two pointers will always point to adjacent nodes so that if we find the node to be deleted we have a pointer to the preceding node. You will recall from our previous work with linked lists that we need a pointer to the preceding node in order to perform the deletion.
- The rest of the procedure sets up the two pointers and moves them down the list until we find the node to be deleted or until we can no longer move down the list. The appropriate action is then taken.

6. When working with linked lists it is very important to consider all of the special cases that need to be handled separately. It is very common for beginning programmers to overlook some of these possibilities.
   - When this happens, the procedure usually commits the run-time error of trying to access some information from a non-existent node. That is, it will try to access data (either info or next) using a pointer that is set to NIL.

7. The dynamic implementation of linked lists has many advantages over the array implementation.
   - The major advantage is that nodes do not have to be created in advance. If our program uses several linked lists, the nodes for each list are created as they are needed. An overflow condition is possible only when all the memory in the computer is used. With the array implementation, we can overflow one array even if there is unused memory in the computer.
   - Another advantage is that of quicker access time. Since a pointer is an address, the computer can get to node P directly by looking at the address stored in P. (With the array implementation, the computer can only get to node (P) by computing the address of the node using the base address of array node, the length of each node and the contents of location P.)
   - A third advantage is that the system handles the allocation and freeing of memory. The programmer does not have to maintain an available list.
   - The only possible disadvantage of the dynamic implementation is that it may be more time-consuming to call upon the system to allocate and free storage than to manipulate a programmer-managed available list.

Summary:

1. What are the three major advantages of the dynamic implementation of linked lists over the array implementation?

2. What is the major advantage of the array implementation of linked lists over the dynamic implementation?

Exercises:

1. Using the declarations from the last lesson, write a procedure, called Printlist (list), which will print out all the elements of the dynamically-implemented linked list "list" starting at the first element.

2. Redo exercise #1 as a recursive procedure.

3. Assume we are working with ordered linked lists. Using the declarations from the last lesson, write a procedure that will insert a node with the value X in its proper place in linked list "list." Be sure you consider all the special cases.

4. Write a program that implements a stack using a dynamically-implemented linked list. The program should input integer X and perform the following actions depending on the value of X:

   - If X is positive then push X, print the entire stack, input another X and continue the process.

   - If X is negative then pop the stack, print the value of the node popped, print the entire stack, input another X, and continue the process.

   - If X is 0 then print out the stack and terminate the program.

   NOTE: Be sure to check for "stack underflow" when popping.

5. Write a program that implements a queue using a dynamically-implemented linked list. It should be exactly like Exercise #4, except that deletions should be performed in FIFO order. You will have to replace push and pop with the corresponding insert and delete procedures.

TOPIC 79-80

AIRLINE RESERVATION SYSTEMS

Performance Objectives - Students will be able to:

- discuss major business applications of linked lists
- explain how top-down design can be effectively applied to a large-scale problem
- show how a linked list can be used to create an airline reservation system
- describe how a dynamically-implemented linked list can be maintained

Motivation:

- What are some of the business uses of linked lists?
  - Mailing lists, accounts receivable, accounts payable, inventory, orders to be filled, reservations, etc.
- Most business programs deal with maintaining lists. These lists are created, modified, sorted, accessed and printed in various ways. It is, therefore, natural for a linked list structure to play an important role in business programs.

Aim: How can an airline reservation system be created using a linked list?

Development:

1. Examine the problem of creating an airline reservation system. Though simplified, our system will contain all the basic features of a full system. Start by stating the requirements of the problem and then solve it in a top-down fashion.

2. The input consists of two types of data: flight data used to initialize the system and passenger data containing the passenger reservation requests to be processed.

   - The flight data consists of a single number (representing the number of flights available that day) followed by the flight number and the seating capacity of each flight.

   - Once the flight data has been read, a separate input line is read for each passenger request for service. The requests may be of three types: reservation, cancellation, or inquiry. The type of each request is indicated by the letter 'R', 'C' or 'I'.

     - A request for a reservation or a cancellation is accompanied by a flight number and a passenger name.

     - An inquiry is accompanied by a passenger name. A list of all flights on which the passenger has reservations is returned.

- For each passenger service request, a message describing the action taken is to be printed.

3. Before continuing, determine the action to be taken for each of the three types of passenger requests under the various possible circumstances.

   - In the case of a reservation, the passenger is placed on the flight's flight list, if it is not full. If it is, the passenger is added to the waiting list.

   NOTE: We do not check if the person already has a reservation on that flight, but this is something that you might want to consider.

   - In the case of a cancellation, the passenger is deleted from the flight list if booked, and the first passenger from the waiting list (if any) moves onto the flight list.

     - If the cancelled passenger is on the waiting list, he must be removed from it.

   - Finally, in the case of an inquiry, a list of all flights on which the passenger is booked is to be printed. It should indicate whether the passenger is on the flight list or waiting list of each flight.

4. The next step involves deciding on the data structures and variables to be used.

   - Design the program to handle up to 100 flights and represent each flight as a record. The entire flight information can be kept in an array of these flight records.

     - Each record contains the flight number, the seating capacity, the number of passengers currently on the flight, the flight list and the waiting list for that flight.

   - The last two items, flight list and waiting list, are maintained as linked lists and the record will contain a pointer for each list.

   - The non-standard Pascal string type will be used for the passenger names and their values will be entered using readln. This was done to simplify the writing of the program. Make any necessary modifications for your implementation.

     - We need the following constant, types and variable:

```
CONST
  maxflights = 100;
TYPE
  passptr = ↑passenger;
  passenger =
    RECORD
      name : string;                (* passenger name *)
      next : passptr                (* next passenger on list *)
    END;
  flighttype =
    RECORD
      fltno : integer;              (* flight number *)
      cpcty : integer;              (* capacity of flight *)
      count : integer;              (* number of passengers *)
      fltlist : passptr;            (* ptr to 1st passenger on
                                    flight list *)
      waitlist : passptr            (* ptr to 1st passenger on
                                    waiting list *)
    END;
VAR
  flight : ARRAY [1..maxflights] OF flighttype;
```

5. We will also need some other global variables.

   - First, we need a variable, called numflights, for the number of flights.

   - We will also need two variables, I and J, for our loops. Finally, we need a variable of type char, called command, to read in the passenger request commands. The need for these variables will soon be evident.

   - Our additional variables are as follows.

```
VAR    (* additional variables *)
  numflights : 0..maxflights;       (* number of flights *)
  I, J : integer;                   (* for loops *)
  command : char;
```

6. The main outline for the program is:

```
readln (numflights);
FOR I := 1 TO numflights DO
  BEGIN
    read in flight data;
    initialize flight [I]
  END; (* FOR *)
WHILE NOT eof DO
  BEGIN
    read command;
    IF command = 'I' THEN
      call inquire procedure
    ELSE
      IF command = 'R' THEN
        call reserve procedure
      ELSE
        IF command = 'C' THEN
          call cancel procedure
        ELSE
          writeln ('Invalid command');
  readln
END (* WHILE *)
```

7. Except for the three procedures introduced, the outline can easily be translated into Pascal.

   - Notice we are only reading the command portion of the passenger requests in the main program. The rest of the request information is read by the appropriate procedure.

8. Write the three procedures: inquire, reserve and cancel. First outline them and then revise them through stepwise refinement.

   - To further simplify the solution, write some procedures and functions that may be useful to the three main procedures.

   - Since we will be treating the waiting list's linked list as a queue, it would be helpful to have a procedure to insert a passenger at the end of the list and another to move the first person on the waiting list to the flight list. Because the flight list does not have to be maintained as a queue, we will insert passengers at the front of the list. This will require a different insert procedure.

   - When passengers are removed from the lists, they may be located anywhere on the lists. We will, therefore, need a procedure to delete a passenger from a list.

   - For the inquire and cancel procedure, we need to know if a name is on a list. We will write a boolean function to determine whether or not a name is on a list.

   - Finally, we will write a function that will return the subscript for a flight (in the array flight) given the flight number.

9. We will now write the headings for the above procedures and functions. A comment follows each heading.

```
FUNCTION Getfltsub (fnumber : integer) : integer;
    (* Searches array "flight" for the subscript of the record for
    flight with flight number "fnumber" and returns the subscript.
    The value 0 is returned if that flight number does not exist *)

FUNCTION Onlist (ptr : passptr; passname : string) : boolean;
    (* Searches the list pointed to by "ptr" for the passenger named
    "passname" and returns a value of TRUE if the name is found, or
    FALSE if it is not. *)

PROCEDURE Rearinsert (VAR ptr : passptr; passname : string);
    (* Inserts the passenger named "passname" onto the end of the list
    "ptr" *)

PROCEDURE Frontinsert (VAR ptr : passptr; passname : string);
    (* Inserts the passenger named "passname" onto the front of the
    list "ptr" *)

PROCEDURE  Delete (VAR ptr : passptr; passname : string);
    (* Deletes the passenger named "passname" from the list "ptr" *)
```

```
PROCEDURE Waittoflight (subscript : integer);
    (* Moves first passenger from the waiting list to flight list on
    the flight with the subscript "subscript" *)
```

11. Write the three main procedures using the above procedures and functions.

    . Of course, we will then have to go back and complete the above six procedures and functions. But notice that since we have the headings and descriptions for these procedures and functions we do not have to wait for them to be completed.

      - In fact, one person may now proceed with the three main procedures while another person works on the other six.

12. Begin with the inquire procedure. First simplify it, as well as the other two procedures, using the technique of stepwise refinement. (We will skip that step here and proceed to the final solution.)

```
PROCEDURE Inquire;
VAR
  I : integer;
  pname : string;
BEGIN
  writeln ('Enter passenger name: ');
  readln (pname);
  writeln ('Inquire about passenger ', pname);
  writeln ('Passenger is on following flights: ');
  FOR I := 1 TO numflights DO
    IF onlist (flight[I].fltlist, pname) THEN
      writeln (flight[I].fltno, '-- on flight')
    ELSE
      IF onlist (flight[I].waitlist, pname) THEN
        writeln (flight[I].fltno, '-- waiting list')
END; (* Inquire *)
```

13. The reserve procedure is next:

```
PROCEDURE Reserve;
VAR
  pname : string;
  I,
  flt,
  sub : integer;
BEGIN
  writeln ('Enter passenger name: ');
  readln (pname);
  writeln ('Enter flight number: ');
  readln (flt);
  writeln ('Reserve on flight ', flt, ' for ', pname);
  sub := getfltsub (flt);
  IF sub = 0 THEN
    writeln ('Invalid flight number.')
  ELSE
```

```
    IF flight[sub].count < flight[sub].cpcty THEN
      BEGIN
        frontinsert (flight[sub].fltlist, pname);
        flight[sub].count := flight[sub].count + 1;
        writeln ('Reservation confirmed.')
      END
    ELSE
      BEGIN
        rearinsert (flight[sub].waitlist, pname);
        writeln ('On waiting list.')
      END
END; (* Reserve *)
```

14. Next is the cancel procedure:

```
PROCEDURE Cancel;
VAR
  pname: string;
  I,
  flt,
  sub : integer;
BEGIN
  writeln ('Enter passenger name: ');
  readln (pname);
  writeln ('Enter flight number: ');
  readln (flt);
  writeln ('Cancel ', pname, ' on flight ', flt);
  sub := getfltsub (flt);
  IF sub = 0 THEN
    writeln ('Invalid flight number')
  ELSE
    IF onlist(flight[sub].fltlist, pname) THEN
      BEGIN
        delete (flight[sub].fltlist, pname);
        IF (flight[sub].count = flight[sub].cpcty) AND
        (flight[sub].waitlist<>NIL) THEN
          waittoflight (sub)
        ELSE
          flight[sub].count := flight[sub].count -1;
        writeln ('Removed from flight list.')
      END
    ELSE
      IF onlist (flight[sub].waitlist, pname) THEN
        BEGIN
          delete (flight[sub].waitlist, pname);
          writeln ('Removed from waiting list.')
        END
    ELSE
      writeln (pname, ' not on flight ', flt)
END;  (* cancel* )
```

15. Now write the entire program. Do not include the nine procedures and functions. The three above procedures and the preceding six, which will be left as exercises, will not be included in our program.

```
PROGRAM Airline (input, output);
CONST
  maxflights = 100;
TYPE
  passptr = ↑passenger;
  passenger =
    RECORD
      name : string;
      next : passptr
    END;
  flighttype =
    RECORD
      fltno,
      cpcty,
      count : integer;
      fltlist,
      waitlist : passptr
    END;
VAR
  flight : ARRAY [1..maxflights] OF flighttype;
  numflights : 0..maxflights;
  I, J : integer;
  command : char;

(* The following procedures and functions should be inserted here:
Getfltsub, Onlist, Rearinsert, Frontinsert, Delete, Waittoflight,
Inquire, Reserve and Cancel *)

BEGIN (* Airline*)
  writeln ('Enter number of flights: ');
  readln (numflights);
  FOR I := 1 TO numflights DO
    BEGIN
      writeln ('Enter flight number: ');
      readln (flight[I].fltno);
      writeln ('Enter capacity: ');
      readln (flight[I].cpcty);
      flight[I].count  := 0;
      flight[I].fltlist  := NIL;
      flight[I].waitlist := NIL
    END;
  WHILE NOT eof DO
    BEGIN
      writeln;
      writeln ('Enter command -- I, R, or C: ');
      readln (command);
      IF command IN ['I', 'R', 'C'] THEN
        CASE command OF
```

```
          'I' : Inquire;
          'R' : Reserve;
          'C' : Cancel
        END
      ELSE
        writeln ('invalid command')
  END (* WHILE *)
END.  (*Airline*)
```

Summary:

1. What are the major business applications of linked lists?

Exercises:

1. Write the Getfltsub, Onlist, Rearinsert, Frontinsert, Delete, and Waittoflight procedures/functions.

2. Modify the airline program to list all the passengers on the flight list and waiting list for a given flight upon reading the command 'F' followed by a flight number.

3. Modify the airline program to accept and process a modify command 'M' with a flight number and a quantity. The quantity represents an increase or decrease in the number of seats available on that flight.

TOPIC 81

BINARY TREES

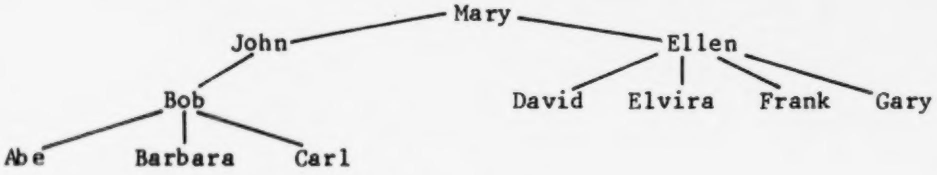
Performance Objectives - Students will be able to:

- define a binary tree recursively
- explain terms related to binary trees
- describe operations that create binary trees and form left and right children to a specified node.
- provide a formula for the number of nodes in a complete binary tree

Vocabulary: binary tree, root, left and right subtree, node, father, son, brother, parent, child, sibling, ancestor, descendant, leaf, level, complete binary tree.

Motivation:

- Draw a diagram to represent the family tree for the following situation. Mary has two children named John and Ellen. John has one child, named Bob, who in turn has three children named Abe, Barbara, and Carl. Ellen has four children named David, Elvira, Frank, and Gary.



Aim: How to create a binary tree.

Development:

1. The above diagram is an example of a simple family tree that shows the children, parents and siblings of the people represented in the tree.

    - Such a representation could also represent the hierarchy of a company so that: One, Mary is the immediate supervisor of John and Ellen. And two, Abe, Barbara and Carl are co-workers who have Bob as their immediate supervisor.

2. The above diagram is a simple example because there are no complicated connections. Viewed as a company hierarchy:

    - Each person can have lower level employees working under his supervision.

    - Each person, except for Mary who is the head of the company, has one and only one immediate supervisor.

    - Also, no one supervises people at the same or higher level (that is, Bob does not supervise David or Ellen or Mary.)

- If any of these relationships existed in the company, the tree representation would become more involved.

3. The above hierarchical representation, which has many uses, is called a 'tree'.

   - A binary tree is a simple tree where no person has more than two employees working under his supervision (or viewed as a family tree, no parent has more than two children).

   - The binary tree is defined recursively. This is done because the recursive definition is a simple way to define a binary tree which includes all of the restrictions alluded to so far.

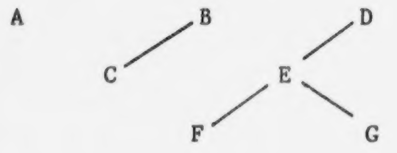
4. A binary tree is a finite set of elements which is either empty or consists of a single element called the root and whose remaining elements are partitioned into two disjoint subsets, each of which is also a binary tree.

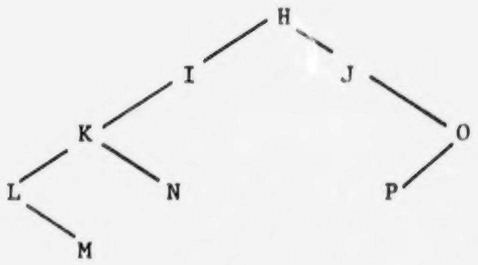
   - The two subsets are referred to as the left subtree and the right subtree of the original tree.

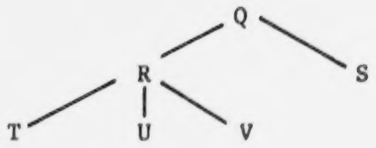
   - Each element of a binary tree is called a node of the tree.

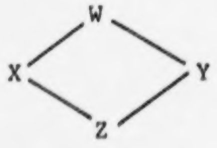
5. The above definition illustrates both the power and confusion that can be created from a recursive definition. In one sentence we defined a binary tree and incorporated all the restrictions that would have taken paragraphs to describe. This "simple" definition eliminates the need of including a long list of conditions that must be met for a structure to be a binary tree.

6. Binary trees are usually represented by diagrams that are similar to the family tree diagram of the motivational problem. Consider the following diagrams:









- The empty set is clearly a binary tree according to the definition.
- The figure which consists of root A is also a binary tree. This is true because we can break up the structure into a root (A) and two disjoint subsets (empty sets) each of which is a binary tree (since we already noted that the empty set is a binary tree).
- The figure with root B is also a binary tree and can be broken up into a root (B), a subset containing C (which is a binary tree by the argument used for A above) and a second subset which is empty, and therefore also a binary tree.
- Similar arguments can be constructed to show that the figures with roots D and H are also binary trees.
- The figure with root Q is not a binary tree because the left subtree with root R breaks down into a root with three disjoint sets.
- The figure with root W is also not a binary tree because it breaks down into a root W and two sets which are not disjoint (since they have Z in common).

7. Use the binary tree with root H to illustrate terms that describe the relationships between the elements. Many of the terms arise from viewing the binary tree as a family tree.
   - Node I is often referred to as the parent or father of node K, thus making node K the child or son of node I.
   - Nodes I and J are often called siblings or brothers since they both have the same parent.
   - Node I is also called the ancestor of node K because node I is the root of a tree which contains node K in one of its subtrees. Node I is, therefore, also the ancestor of nodes L, N and M.
   - Node M is called a descendant of node L because it is contained in a subtree which has node L as a root. Therefore, node M is also a descendant of nodes K, I, and H.
   - Nodes M, N, and P are called leafs because they have no children (that is, both their subtrees are empty).
   - The final term to be introduced is that of level. It can be defined as follows: The root of a tree has level 0 and the level of any node is one more than its parent.
     - Therefore, in our tree node H is of level 0, nodes I and J are of level 1, nodes K and O are of level 2, nodes L, N and P are of level 3, and node M is of level 4.
     - The highest level of all nodes in a tree is called the level of the tree. Our tree with root H would then be of level 4.

NOTE: Some authors define the level of the root as being 1 instead of 0. Be sure you know which definition is being used. In our lessons we will use level 0 as the level of the root.

8. One special type of binary tree is called a complete binary tree of level N. This is basically a binary tree which is full. In particular, each node of level N is a leaf and each node which has a level less than N has nonempty left and right subtrees.

   . A complete binary tree of level 0 will have one node. A complete binary tree of level 1 will have three nodes (one at level 0 and two at level 1). A complete binary tree of level 2 will have seven nodes (one at level 0, two at level 1, and four at level 2).

     - In general, for a complete binary tree of level N: The number of nodes at level K, where K<= N, is 2 raised to the power of K. For example, level 3 contains eight nodes.

   . The total number of nodes in the entire tree can be calculated by adding up the number of nodes in each level. This can easily be done using the formula for the sum of a geometric progression.

     - The result is that the number of nodes is one less than 2 raised to the power of (N + 1). For example, a complete binary tree of level 5 contains 63 nodes. We get this by subtracting 1 from 2 raised to the 6th power.

9. Consider a binary tree as being made up of nodes which have three fields each. To reference the three fields we use the expressions info(node), left(node) and right(node).

   . Info(node) refers to the information that is contained in the node.

   . Left(node) and right(node) reference pointers to the roots of the left and right subtrees of the binary tree with the root named node.

     - These pointers take on a value of NIL when the corresponding subtree is empty.

10. To create a binary tree we will write the operations maketree, setleft and setright in pseudo-code for now and implement them in Pascal later.

    . The function Maketree(X) creates a binary tree consisting of one node and returns (takes on the value of) a pointer to that node. Furthermore, the information field of that node will contain the value X. An algorithm that performs this function is:

```
P :=getnode;
info(P) := X;
left(P) := NIL;
right(P) := NIL;
maketree := P;
```

- Note that P is a temporary pointer and getnode is the function we used when we worked with linked lists.

11. Setleft(P, X) accepts a pointer to a binary tree node with no left child. It creates a new left child of node(P) with a value of X in its information field. An algorithm that will do this is:

```
IF left(P)<> NIL THEN
  writeln ('ERROR--Illegal Setleft')
ELSE
  BEGIN
    Q := maketree(X);
    left(P) := Q
  END;
```

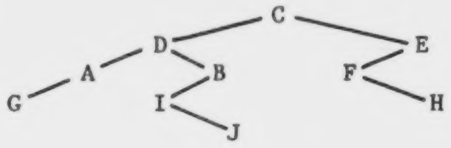
- Setright (P, X) operates in much the same manner, creating a right child with value X.

Summary:

1. What is the recursive definition of a binary tree?
2. What is meant by the terms: root, subtree, node, parent (father), child (son), sibling (brother), ancestor, descendant, leaf, level and complete binary tree?
3. What notation is used for the three parts of a node?
4. What do the operations maketree, setleft and setright do?
5. How many nodes are there in a complete binary tree of level K?

Exercises:

1. For the tree illustrated on the right:
   - What is the root of the left subtree?
   - List all the children of A.
   - List all the the parents of A.
   - List all the siblings of A.
   - List all the the descendants of A.
   - List all the ancestors of A.
   - What is the level of B?
   - What is the level of the tree?



2. How many nodes are in a complete binary tree of level 6? of level 10?
3. In a complete binary tree of level 7, how many nodes are there of level 5? of level 2?

TOPIC 82

BINARY TREE SORT

Performance Objectives - Students will be able to:

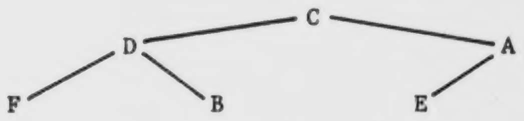
- discuss a technique to organize numerical data in a binary tree
- traverse a binary tree using the inorder method
- describe an algorithm for sorting numbers using a binary tree

Vocabulary: ordered binary tree, traverse, inorder traversal, binary tree sort

Motivation:

- Draw a diagram of the binary tree created by the following statements:

```
root := maketree ('C');
setleft (root, 'D');
setright (root, 'A');
setleft (right(root), 'E');
setright (left(root), 'B');
setleft (left(root), 'F');
```



NOTE: In the above binary tree there is also a pointer named root (not shown) that points to the root of the tree. The root node contains the value 'C'.

Aim: How to use a binary tree for sorting.

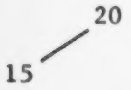
Development:

1. Create a binary tree by first creating the root of the tree and then adding one node at a time. The nodes will contain the values 20, 15, 17, 30, 25, 27, 16, 5, and 40.

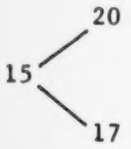
   - The nodes should be added with the above values in the order that the numbers are listed.
   - In addition, a special relationship should exist after all the numbers have been added to the tree.
     - The left subtree of each node should contain only values that are less than the value of that node.
     - The right subtree of each node should contain only values which are greater than or equal to the value of that node.
   - Such a binary tree is called an ordered binary tree.

2. Create the binary tree by first creating the root of the tree with a value of 20.

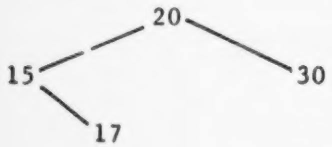
   - Next, add a node with the value of 15. Since 15 is less than 20, this node must be added as the left child of the root. The tree thus far:



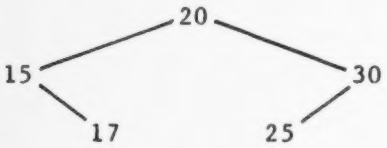
   - Now, add a node with the value of 17. Since it must belong to the left subtree of 20 and the left child of 20 already exists, go to another level deeper in the left subtree. Because 17 is greater than 15, add 17 as the right child of 15. Examine the following diagram and check to see that the conditions are still met.



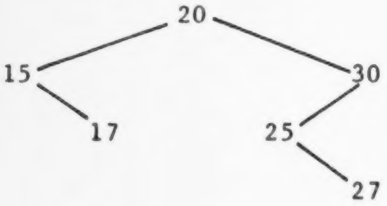
   - Now add a node with a value of 30 to the tree. Since 30 is greater than 20, add it as the right child of node 20. This gives us:



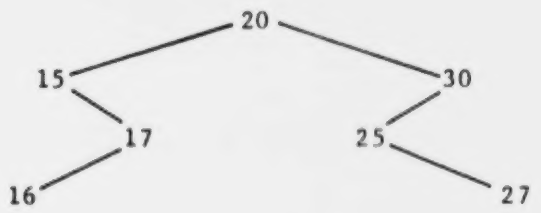
   - The next value to be added is 25. Since it is greater than 20, it must be contained in the right subtree. The root of that subtree has a value of 30, so the new node must be contained in its left subtree since 25 is less than 30. We now have:



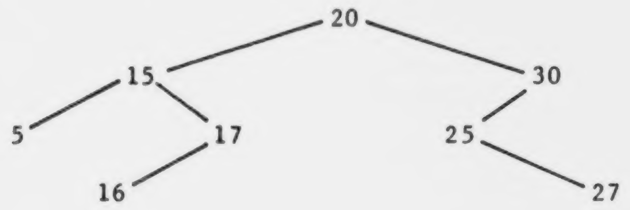
   - Next add the value of 27 in a similar manner. This gives us:



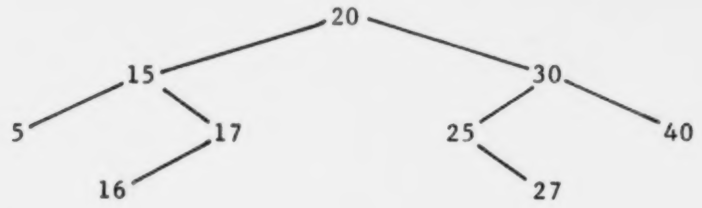
- Adding the value of 16 yields:



- Add 5 to the tree to get:



- Finally, add the last value, which is 40, to the tree.



- Examine this tree and verify that it is indeed an ordered binary tree. Check the left and right subtree of each node.

NOTE: If there were repeated numbers, they would have gone in the right subtree. This is because we had the values that were greater than or equal to the root value placed in the right subtree.

- An ordered binary tree can also be created where the values in the left subtree are less than or equal to the root value and the values in the right subtree are the greater values.

3. We will now examine one way to traverse the tree. To traverse a tree means to pass through the tree, operating on each node once. We will operate on each node by printing out its value. Our method of traversal must insure that each node is operated on exactly once. The method we shall choose is called inorder traversal and we will define it recursively.

- To traverse a nonempty binary tree in inorder, perform these steps:

  1. Traverse the left subtree in inorder.
  2. Operate on the root.
  3. Traverse the right subtree in inorder.

NOTE: You do not have to do anything to traverse an empty binary tree.

4. Try to perform an inorder traversal of the ordered binary tree above, the one which has a root value of 20.

   - Begin by traversing the left subtree using the inorder procedure. That is the tree which contains 15, 5, 17 and 16.

     - Again, according to the procedure we must first traverse the left subtree of this tree with root value 15. That becomes the tree containing only the value 5.

     - Once more, we first traverse the left subtree. Since its left subtree is empty, we proceed to operate on the root whose value is 5. To do so, we merely print out its value (5).

     - Next, we traverse the right subtree of (the root with value of) 5. Since this subtree is also empty, there is nothing to do.

     - We have now completed the traversal of the left subtree of 15. We print out the value of the root of that tree, which is 15 (so far we have printed out 5 and 15).

     - Next, we traverse the right subtree with root value 17. To traverse this tree we must first traverse its left subtree with root value 16.

     - Since the left subtree of 16 is empty, we print out its value (so far we have printed out 5, 15 and 16). The right subtree of 16 is also empty, so we have completed the traversal of the left subtree 17 and can now print out the value of 17 (so far we have printed out 5, 15, 16 and 17).

   - Since the right subtree of 17 is empty, this concludes the traversal of the right subtree of 15 which, in turn, concludes the traversal of the left subtree of 20.

   - We next print out the value of 20 which makes the output so far: 5, 15, 16, 17 and 20.

   - We then proceed to output the right subtree of 20 in similar fashion. After the entire tree has been traversed, the output should be as follows: 5, 15, 16, 17, 20, 25, 27, 30 and 40.

5. Observe that the values in our tree were printed out in numerical order. This will be true any time we print out an ordered binary tree using the inorder traversal method.

   - We can print out a non-ordered binary tree with the inorder method of traversal, but the values will not be in numerical order.

   - If the node values of the ordered binary tree were strings, then inorder traversal would output those strings in alphabetical order.

6. In the previous paragraphs we have taken a list of numbers, which were not in order, and printed them out in numerical order. In other words, we have sorted the numbers. The process used is the basis of a method known as a <u>binary tree sort</u>. To sort numbers (or strings) we place them into an ordered binary tree and then traverse the tree using the inorder method.

7. Up to now we have only illustrated the method of using a list of given values to create an ordered binary tree. We will now give an algorithm, using the operations from the last lesson, that performs this task.

```
readln (X);
root := maketree (X);
WHILE NOT eof DO
  BEGIN
    readln (X);
    P := root;
                  (* Set P to point to the parent of the new node *)
    WHILE ((X < info(P)) AND (left(P)<>NIL))  OR
    ((X>= info(P)) AND (right(P)<>NIL)) DO
      IF X< info(P) THEN
        P := left(P)
      ELSE
        P := right(P);
                 (* Create new node with value X in proper location *)
    IF X< info(P) THEN
      setleft (P, X)
    ELSE
      setright (P, X)
  END;
```

<u>Summary:</u>

1. What relationship exists between the root values and the subtree values in an ordered binary tree?

2. How can we create an ordered binary tree?

3. What do we mean by the term "traverse a tree"?

4. How do we traverse a binary tree using the inorder method?

5. How can a binary tree be used to sort a list of values?

<u>Exercises:</u>

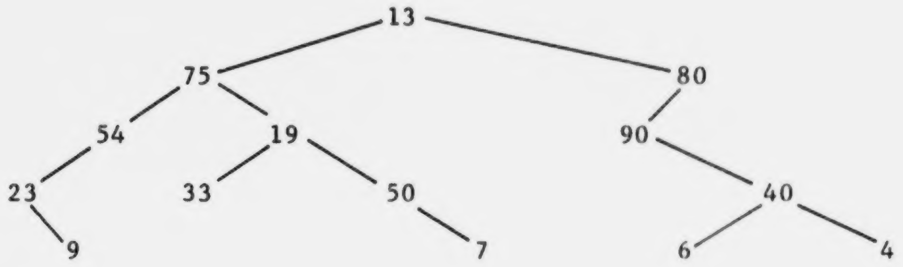
1. Draw the binary tree that is created by the following set of statements:

```
tree := maketree (15);
setright (tree, 9);
setright (right(tree), 3);
setleft (tree, 12);
setright (left(tree), 7);
setleft (right(right(tree)), 10);
```

2. Show the ordered binary tree that is created when we use our algorithm to input the following values: 7, 10, 8, 4, 6, 6, 7, 9, 5, 5 and 4.

3. Show the ordered binary tree that is created when we use our algorithm to input the following string values: 'John', 'Tom', 'Betty', 'Dave', 'Connie', 'Cynthia', 'Ron', 'Larry', and 'Donna'.

4. Give the output if the following tree is printed out using our inorder traversal procedure. Note that the tree is not an ordered tree.



TOPIC 83

BINARY TREE TRAVERSAL

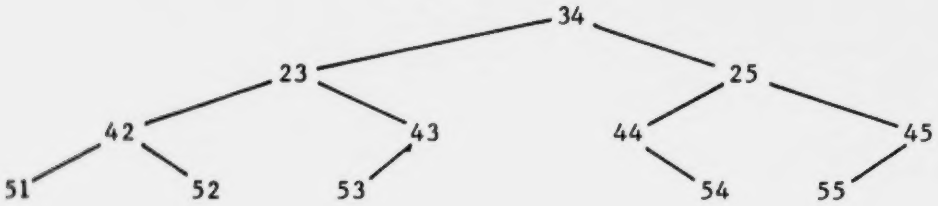
Performance Objectives - Students will be able to:

- traverse a binary tree in preorder, inorder and postorder
- form binary tree representations of algebraic expressions that contain infix operators
- demonstrate how traversals of binary tree expressions produce prefix, infix and postfix forms of that expression

Vocabulary: preorder, inorder and postorder traversal

Motivation:

- Give the output if the following tree is printed out using the inorder method of traversal.



- The output will be the following: 51, 42, 52, 23, 53, 43, 34, 44, 54, 25, 55 and 45.

Aim: How to traverse a binary tree.

Development:

1. In the last lesson we recursively defined the method of inorder traversal as follows:

   - To traverse a nonempty binary tree in inorder, perform these steps:

     1) Traverse the left subtree in inorder.
     2) Operate on the root.
     3) Traverse the right subtree in inorder.

2. When we traverse an ordered binary tree using the inorder method, the nodes are operated on in numerical (or alphabetical) order. While this method of traversal is very important and often used, there are other traversal methods that perform equally important tasks for other types of binary trees.

- In this lesson we will learn two other methods of traversal that are very similar to the inorder method. We will also examine some applications of tree traversals.

3. Our first new traversal is called a preorder traversal. When we performed an inorder traversal we operated on the root between the traversals of the subtrees. In the preorder traversal we operate on the root before we traverse the subtrees. This leads to the following recursive definition.

   - To traverse a nonempty binary tree in preorder:

     1) Operate on the root.
     2) Traverse the left subtree in preorder.
     3) Traverse the right subtree in preorder.

4. To illustrate this traversal print out the binary tree from the motivational problem using the preorder method.

   - First print out the root value 34 and then begin to traverse the left subtree whose root value is 23. Print this root value of 23 and traverse its left subtree (with root value 42). This causes the root value of 42 to get printed and we proceed to its left subtree (with root value 51). We then print out the value 51. So far we have printed 34, 23, 42, and 51.

   - Since the tree whose root value is 51 has empty left and right subtrees, its traversal is completed. Now proceed to the right subtree of 42, which is 52. Printing out the value of 52 completes the traversal of both the tree with root value of 52 and the left subtree of 23.

   - Our next tree is the right subtree of 23 which has a root value of 43. So we print out the value of 43 and then traverse its left subtree whose root is 53. The value of 53 is printed next and we proceed to traverse its subtrees. Since both of its subtrees are empty, we have completed the traversal of the tree with root 53 (that is, the left subtree of 43).

     - Since 43's right subtree is empty, its traversal is complete. Because 43 is the right subtree of 23, we have also completed the traversal of the tree with root value 23 which is the left subtree of 34.

   - To complete the entire traversal of the original tree you must next traverse the right subtree of 34. When this has been completed the output should be: 34, 23, 42, 51, 52, 43, 53, 25, 44, 54, 45 and 55.

5. The other new traversal is called a postorder traversal. In the postorder traversal we operate on the root after we traverse the subtrees. The following recursive definition describes the process.

   - To traverse a nonempty binary tree in postorder:

     1) Traverse the left subtree in postorder.
     2) Traverse the right subtree in postorder.
     3) Operate on the root.

6. We will use the same binary tree to illustrate this method of traversal.

   - First traverse the left subtree of 34 whose root value is 23. To do this, we traverse its left subtree whose root value is 42 and this in turn requires traversing its left subtree whose root value is 51.

     - To traverse this tree we must first traverse its left subtree. Since it is empty, we proceed to the right subtree of 51, which is also empty. Because both of 51's subtrees are empty, we then print the root value of 51. This completes the left subtree of 42.

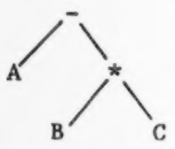
     - Next traverse the right subtree of 42, the one whose root value is 52. Since both of its subtrees are empty, we print the value of 52. This completes the traversal of both subtrees of 42 which means we can now print the value of 42. This completes the left subtree of 23.

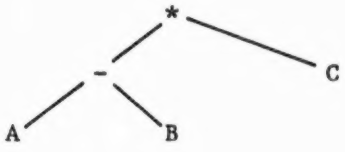
   - To complete the entire traversal of the original tree, traverse the right subtree of 23, repeating the above process until you can print out the root value of 23. Then, traverse the right subtree of 34, repeating the process until the entire tree has been output. The complete output should be: 51, 52, 42, 53, 43, 23, 54, 44, 55, 45, 25 and 34.

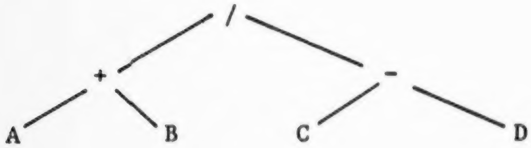
7. Now look at an application which uses binary trees to represent algebraic expressions involving operands and binary operators for addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (for which we introduce the symbol∧).

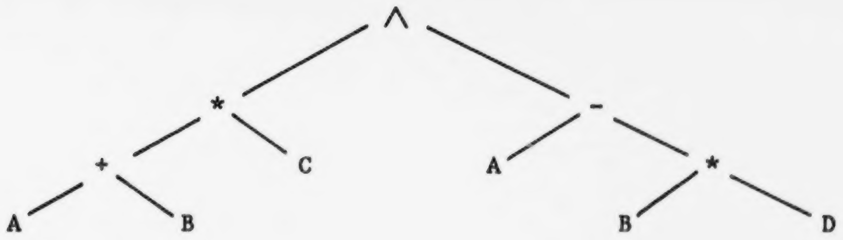
   - The operands and binary operators will be contained in the nodes of the binary tree. When a binary operator is a root of a tree (or subtree) it is considered to have the left and right subtrees as its operands. Since our operations are binary, a binary operator node will never have an empty subtree. Also, all operand nodes will always have empty left and right subtrees.

   - The following diagrams illustrate the binary tree representation of the following algebraic expressions: A - B * C, (A - B) * C, (A + B) / (C - D), and ((A + B) * C) ∧ (A - B * D).









8. In Topic 67 we saw that there were ways to write an algebraic expression other than our usual infix method. Recall that in our infix notation the binary operator is written between its operands. The binary operator could also be placed before or after its two operands producing the prefix and postfix notations, respectively.

   . Take a binary tree that represents an algebraic expression and examine its printout using the three methods of tree traversal. If we print out this tree using the preorder method, the binary operator will precede its operands (subtrees). By examining the results you will see that this preorder traversal produces the prefix form of the expression.

   . Likewise, a printout using the postorder method of traversal produces the equivalent postfix form.

   . Unfortunately, when we print out the tree using the inorder method of traversal we do not obtain a true infix form. The output will be the infix form of the expression without parentheses. If we try to evaluate the expression using our usual order of operation we will not always obtain an equivalent expression.

   . To illustrate the printout of binary tree expressions, examine the three forms of output for the four trees of the last paragraph.

9. The preorder traversal of these trees produces the following four prefix expressions:

   - A * B C
   * - A B C
   / + A B - C D
   ∧ * + A B C - A * B D

   . The postorder traversal yields the following postfix expressions:

   A B C * -
   A B - C *
   A B + C D - /
   A B + C * A B D * - ∧

   . The inorder traversal produces expressions that are almost infix equivalents of the expressions represented. They are correct except for parentheses. The outputs are:

A - B * C
A - B * C
A + B / C - D
A + B * C ∧ A - B * D

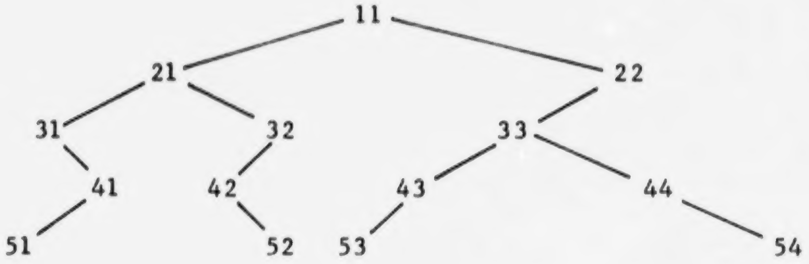
Summary:

1. What are three methods of traversing a binary tree? Describe each.

2. What do we get when we traverse a binary tree representation of an algebraic expression using our three methods of traversal?

Exercises:

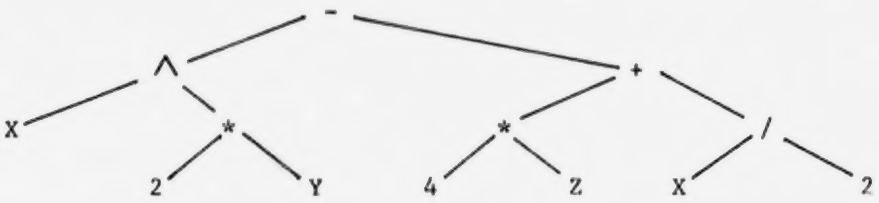
1. Determine the inorder, preorder and postorder printouts for the following binary tree.



2. Draw the binary tree representations of the following algebraic expressions which are written in infix form:

   X - Y * (A - B)
   A + B - C + D ∧ 2
   X ∧ 2 + Y ∧ 3 / Z - 2 * X * Y

3. Write the prefix, postfix and infix forms of the algebraic expression represented in the following binary tree. Be sure to include all necessary parentheses for the infix form.



TOPIC 84

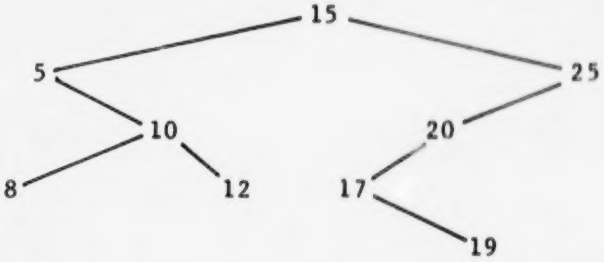
ARRAY IMPLEMENTATION OF BINARY TREES

Performance Objectives - Students will be able to:

- write a declaration that implements binary tree nodes using array elements
- create subprograms for common tree operations using the array implementation
- traverse a binary tree using an array implementation

Motivation:

- Draw a diagram showing the tree that is created when the following values are put into an ordered binary tree: 15, 5, 25, 10, 20, 8, 17, 12 and 19.



- Show the output if the tree is printed out using each of the three methods from the last lesson.

```
Inorder:    5,  8, 10, 12, 15, 17, 19, 20 and 25
Preorder:  15,  5, 10,  8, 12, 25, 20, 17 and 19
Postorder:  8, 12, 10,  5, 19, 17, 20, 25 and 15
```

Aim: How to implement binary trees in Pascal using arrays.

Development:

1. In the last three lessons our work with binary trees has only been through illustrations, descriptions and pseudo-Pascal operations. We will now implement them in Pascal.
   - In this lesson we will use an array of records for the implementation, while in the next lesson we will use pointer variables to create a dynamic implementation.
   - This two step process is very similar to the way we developed linear linked lists. In fact, many of the operations are taken from our work with linked lists. We will rush through much of the duplicated work and ask you to go back to the lessons on linked lists if you want to see the full development of these concepts.

2. Our nodes will be records, which contain an integer info value and two pointers, left and right, which point to the left and right subtrees. The tree will be made up of records of this type and stored in an array called node. As in our array implementation of linked lists, our pointers will be subscripts to the array. Again, the pointer value of 0 will denote NIL. A TYPE and VAR declaration similar to the following can be used.

```
TYPE
  nodeptr = 0..100;
  nodetype =
    RECORD
      info : integer;
      left, right : nodeptr
    END;
VAR
  node : ARRAY [1..100] OF nodetype;
  root,
  avail : nodeptr;
```

3. In the above declaration we declared two pointer variables called root and avail. The variable root points to the root node of our binary tree. The variable avail points to the linear linked list of available nodes.

   - Note that even though we are working with tree nodes our available nodes will be maintained in a linear linked list. We will again treat this available list as a stack. We will use the right pointer of each node to connect the nodes in our available list.

   - To see how this list of available nodes is initialized and maintained, you can look back at our work with lists or read through the procedure called initialize and the function called getnode in the program at the end of this lesson. You will see that this getnode function is similar to the one we had with lists.

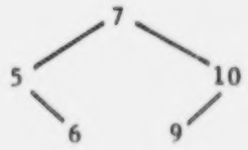
   - A freenode procedure, like the one with lists, is left as an exercise.

4. To illustrate this representation, the following binary tree is shown along with its array implementation. Only the significant portion of the array is shown.

NODE

	info	left	right
1	5	0	3
2	10	5	0
3	6	0	0
4	7	1	2
5	9	0	0
6			7
7			8

avail = 6
root = 4



5. Using our new notation, the terms info(P), left(P) and right(P) are referenced by the terms node[P].info, node[P].left, and node[P].right, respectively.

   - With these new terms, we can implement the maketree, setleft and setright operations. These operations are included in the program at the end.

   - Notice how the rewriting of these operations is nothing more than a copy of the previously written operations with our new notation for info(P), left(P) and right(P).

6. Since we define trees recursively, most of the programs and procedures written for trees are recursive. This is almost always true for recursively-stated procedures, such as inorder traversal.

   - A very simple and straightforward recursive procedure that outputs the values of a binary tree using the inorder method follows directly from the definition of inorder traversal. This procedure is also included in the following program.

7. Any program or procedure written with our pseudo-Pascal operations can easily be modified to work with a Pascal array implementation. As an example, we will rewrite many of the operations from the last few lessons and include them in the following program which reads integer values into an ordered binary tree and then outputs the tree using the inorder method. (Note: For greater flexibility, variable parameters can be added to some of the following subprograms.)

```
PROGRAM Obtree (input, output);
CONST
  numnodes = 100;
TYPE
  nodeptr = 0..numnodes;
  nodetype =
    RECORD
      info : integer;
      left,
      right : nodeptr
    END;
VAR
  node : ARRAY [1..numnodes] OF nodetype;
  avail,
  root : nodeptr;

PROCEDURE Initialize; (* Initializes avail and root and sets up linked
          list of available nodes. *)
VAR
  N : nodeptr;
BEGIN
  root := 0;
  avail := 1;
  FOR N := 1 TO numnodes - 1 DO
```

```
    node[N].right := N + 1;
  node[numnodes].right := 0
END; ( * Initialize *)

FUNCTION  Getnode : nodeptr; (*Returns a pointer to the next available
          node *)
BEGIN
  IF avail = 0 THEN
    writeln ('Error - Overflow')
  ELSE
    BEGIN
      getnode := avail;
      avail := node[avail].right
    END
END; (* Getnode *)

FUNCTION  Maketree (X : integer) : nodeptr; (* Returns a pointer to a
          newly created binary tree whose single node has value X *)
VAR
  P : nodeptr;
BEGIN
  P := getnode;
  node[P].info := X;
  node[P].left := 0;
  node[P].right:= 0;
  maketree := P
END; (* Maketree *)

PROCEDURE Setleft (P : nodeptr; X : integer); (* Creates a node with
          value X as the left child of node P. Assume P is not NIL *)
VAR
  Q : nodeptr;
BEGIN
  IF node[P].left<>0 THEN
    writeln ('Error - Illegal Setleft')
  ELSE
    BEGIN
      Q := maketree(X);
      node[P].left := Q
    END
END;(* Setleft *)

PROCEDURE Setright(P : nodeptr ; X : integer); (* Creates a node with
          value X as the right child of node P.  Assume P is not NIL *)
VAR
  Q : nodeptr;
BEGIN
  IF node[P].right<>0 THEN
    writeln (' Error - Illegal Setright')
  ELSE
    BEGIN
      Q := maketree(X);
      node[P].right := Q
    END
END; (* Setright *)
```

```
PROCEDURE Create; (* Reads and stores integer values in an ordered
          binary tree.  Root points to the root node *)
VAR
  X : integer;
  P : nodeptr;
BEGIN
  writeln ('Enter integer values for ordered binary tree: ');
  readln (X);
  root := maketree(X);
  WHILE NOT eof DO
    BEGIN
      readln (X);
      P := root;
      WHILE ((X< node[P].info) AND (node[P].left<>0)) OR
      ((X>= node[P].info) AND (node[P].right<>0)) DO
        IF X< node[P].info THEN
          P := node[P].left
        ELSE
          P := node[P].right;
      IF X< node[P].info THEN
        setleft (P, X)
      ELSE
        setright(P, X)
    END
END; (* Create *)

PROCEDURE Inorder (root : nodeptr); (* Outputs a binary tree in
          inorder. Assume root is not NIL *)
BEGIN
  IF node[root].left<>0 THEN
    inorder(node[root].left);
  write (node[root].info, ' ');
  IF node[root].right<> 0 THEN
    inorder (node[root].right)
END; (* Inorder *)

BEGIN (* Main Program *)
  initialize;
  create;
  writeln;
  writeln('The values printed out using inorder traversal are: ');
  inorder (root) ;
  writeln
END.
```

Summary:

1. How do we implement binary trees in Pascal using arrays?

Exercises:

1. Given the following variable and array, draw a diagram of the binary tree it represents.

NODE

root = 3

	info	left	right
1	2	0	0
2	4	0	0
3	9	5	2
4	3	1	6
5	15	7	4
6	7	8	0
7	6	0	0
8	10	0	0

2. Write a procedure to implement the freenode operation. The call freenode(P) should return the node pointed to by P to the front of the available list with pointer avail. You may assume that P is not NIL.

3. Write a procedure to output a binary tree using the preorder method of traversal.

4. For a tree pointed to by the pointer root, write a procedure that outputs the number of nodes in the tree, the sum of all the node values (integers) of the tree and the average node value.

   NOTE: Any of the traversal methods may be used for this type of problem.

TOPIC 85

DYNAMIC IMPLEMENTATION OF BINARY TREES

Performance Objectives - Students will be able to:

- declare array nodes using dynamic variables
- write programs that use the dynamic implementation of binary trees to create children and traverse the tree
- compare the array and dynamic implementation of binary trees

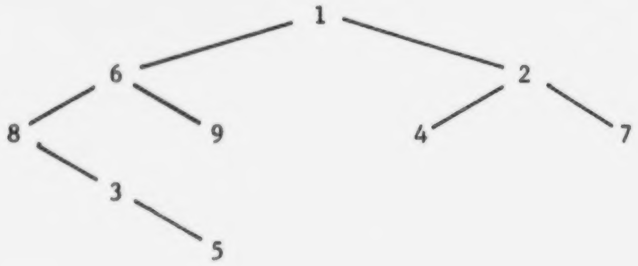
Motivation:

- Given the following variable and array, draw a diagram of the binary tree it represents.

NODE

	info	left	right
1	4	0	0
2	7	0	0
3	6	4	8
4	8	0	6
5	1	3	9
6	3	0	7
7	5	0	0
8	9	0	0
9	2	1	2

root = 5



Aim: How to implement binary trees in Pascal using pointer variables.

Development:

1. In this lesson we will implement binary trees using pointer variables. The method is almost the same as the technique we used to create linked lists using pointer variables. Since this method is dynamic, we need not worry about overflow unless we exceed the memory of the computer.

2. Our nodes will be records with the same three fields used with our array implementation. A TYPE and VAR declaration like the following can be used.

```
TYPE
  nodeptr =↑nodetype;
  nodetype =
    RECORD
      info : integer;
      left,
      right : nodeptr
    END;
VAR
  root : nodeptr;
```

3. The pointer variable root points to the binary tree's root node.

   <u>**NOTE**</u>: No records are created when the program begins execution. Since this method is dynamic, records are created for nodes as they are needed. Also, the computer creates and maintains a list of available nodes, freeing us from this task.

   . Furthermore, the getnode and freenode operations can be replaced with the new and dispose operations. That is, P := getnode is replaced by new(P) and freenode(P) is replaced by dispose(P).

4. Once again, we need to change our terminology for the various parts of a node. The references info(P), left(P) and right(P) will now be P↑.info, P↑.left and P↑.right, respectively.

   . Using these new terms and replacing the empty pointer value of 0 by NIL, rewrite the previous operations so they will work with the pointer variable implementation. The rewritten maketree, setleft and setright operations are included in the program at the end of the lesson.

5. All of the previous programs and procedures can be easily modified for the pointer implementation. As examples, the create and inorder procedures from the last lesson have been rewritten and are included in the program that follows. This program is a direct translation of the program from the last lesson. By observing the changes that were made, you should be able to translate a program from one implementation to another with no difficulty.

```
PROGRAM Obtree2;  (* Reads integer values into an ordered binary tree
                  and then prints using the inorder method *)
TYPE
  nodeptr =↑nodetype;
  nodetype =
    RECORD
      info : integer;
      left,
      right : nodeptr
    END;
VAR
  root : nodeptr;
```

```
FUNCTION  Maketree (X : integer) : nodeptr; (* Returns a pointer to a
          newly created binary tree whose single node has value X *)
VAR
  P : nodeptr;
BEGIN
  new (P);
  P↑.info := X;
  P↑.left := NIL;
  P↑.right :=NIL;
  maketree := P
END; (* Maketree *)

PROCEDURE Setleft (P : nodeptr; X : integer); (* Creates a node with
          value X as the left child of node P.  Assume P is not NIL *)
VAR
  Q : nodeptr;
BEGIN
  IF P↑.left<>NIL THEN
    writeln ('Error - Illegal Setleft')
  ELSE
    BEGIN
      Q := maketree (X);
      P↑.left := Q
    END
END; (* Setleft *)

PROCEDURE Setright (P : nodeptr; X : integer); (* Creates a node with
          value X as the right child of node P.  Assume P is not NIL *)
VAR
  Q : nodeptr;
BEGIN
  IF P↑.right<>NIL THEN
    writeln ('Error - Illegal Setright')
  ELSE
    BEGIN
      Q := maketree (X);
      P↑.right := Q
    END
END; (* Setright *)

PROCEDURE Create; (* Reads and stores integer values in an ordered
          binary tree. Root points to the root node *)
VAR
  X : integer;
  P : nodeptr;
BEGIN
  writeln ('Enter integer values for ordered binary tree: ');
  readln (X);
  root := maketree (X);
  WHILE NOT eof DO
    BEGIN
      readln (X);
      P := root;
```

```
   WHILE ((X < P↑.info) AND (P↑.left<> NIL))  OR
   ((X>= P↑.info) AND (P↑.right<> NIL)) DO
    IF X < P↑.info THEN
     P := P↑.left
    ELSE
     P := P↑.right;
   IF X< P↑.info THEN
    setleft (P, X)
   ELSE
    setright (P, X)
  END
END; (* Create *)

PROCEDURE Inorder (root :nodeptr); (* Prints a binary tree inorder;
     root points to the root node of the tree; assume root is not
     NIL *)
BEGIN
 IF root↑.left<>NIL THEN
  inorder(root↑.left);
 write (root↑.info, ' ');
 IF root↑.right<>NIL THEN
  inorder (root↑.right)
END; (* Inorder *)

BEGIN (* Main Program *)
 create;
 writeln;
 writeln ('The values printed out using inorder traversal are: ');
 inorder (root);
 writeln
END.
```

Summary:

1. How do we implement binary trees in Pascal using pointer variables?

2. What advantages does the dynamic implementation have over the array implementation?

Exercises:

1. Write a procedure to print out a binary tree using the postorder method of traversal.

2. Tell what statements must be changed in Program Obtree2 above so that it reads in string values and alphabetizes them.

3. Write a procedure that traverses a binary tree and prints out the number of node values that are greater than the root value, less than the root value and equal to the root value.

TOPIC 86

HUFFMAN ALGORITHM

Performance Objectives - Students will be able to:

- describe how a message can be encoded into binary bits
- state the Huffman Algorithm that uses a binary tree to minimize the bit code for a message
- demonstrate how a message can be efficiently coded

Vocabulary: Huffman Algorithm, Huffman Tree, strictly binary tree

Motivation:

- We want to store messages in memory using values of 0 and 1 for each bit of memory used. The messages will be a string consisting only of the following five symbols: A, B, C, D and E.

- Devise a code (where each of the symbols is represented by a sequence of 0's and 1's) that minimizes the number of bits needed to store the message 'DABBCAEAEBAA'. Also determine the number of bits needed.

  - Since each character is usually stored in a separate byte of memory, the above string would normally be stored in 12 bytes (96 bits) of memory.

  - However, by using the 7-bit ASCII code (where each character is represented by a sequence of seven bits) for each character, one after the other, our message can be coded with 84 bits.

  - Another improvement would be to use a 5-bit code where each value of the code is 0, except for a single value of 1 in one of the five positions. We could use '10000' to represent A, '01000' to represent B, '00100' to represent C, '00010' to represent D, and '00001' to represent E. We could then code our message using the following 60 bits:

    '000101000001000010000010010000000011000000001010001000010000'

- The previous code used only one significant bit (the one with the value 1) of the 5-bit code. This technique is not very efficient. The 7-bit ASCII code can be used to represent 128 (2 to the 7th power) different characters with its seven bits because it uses each bit position fully. We know that, by using each position fully, an N-bit code can encode "2 raised to the Nth power" different symbols. Since our message contains five different symbols, we need a 3-bit code, which can handle up to eight different symbols. We can simply use '001' for A, '010' for B, '011' for C, '100' for D and '101' for E. We can now encode our message into a sequence of 36 bits as follows:

  '100001010010011001101001101010001001'

  - It may appear that 36 is our minimum number, but this is not true. Consider a variable bit code where we can use '0' for A, '10' for B, '1110' for C, '1111' for D, '110' for E. Using this code our message could be encoded, using only 25 bits, as follows:

'1111010101110011001101000'.

  - At first, it may be difficult to determine when one symbol code ends and the next one begins, but look at the encoding more carefully. If the first bit of a code is 0, the symbol is an A, otherwise it is a B, C, D, or E. In these cases the next bit is examined. If this second bit is a 0, these two bits represent a B, otherwise they are the first two bits of a C, D, or E. If the third bit is a 0 then the three bits make up the symbol E; if not, then they are the first three bits of the 4-bit code for C or D. An examination of the fourth bit will determine whether the four bits are the code for C or D.

- It turns out that a variable bit code using 25 bits produces a coding with the minimum number of bits. There are other variable bit codes which also produce this minimum value, but there are no codes that will improve on this value. In this lesson we will examine the method that produces such a minimum code.

Aim: How to minimize code using a binary tree.

Development:

1. We were able to improve upon the 3-bit code described above because we used a shorter code for symbols, like A, which appeared more often in our message and a longer code for symbols, like C and D, which appeared less often. If we did not know the relative frequencies of the different symbols, then we would not, in general, be able to produce a code that does better than a fixed bit code which uses the least number of bits (for example, the 3-bit code for our motivational problem).

2. While the actual frequencies of the symbols can be determined from the single message or group of messages our code is supposed to encode in a minimum number of bits, this determination is not always possible.

   - For example, we may want to devise a code for a given set of symbols, such as the alphabet, without knowing the message to be encoded. In this case we must use what is known about the relative frequencies of the symbols as they pertain to the messages that will be encoded.

     - If our messages are English words, we could obtain statistics that express the relative frequencies of the various letters in English words.

   - In our examples we will devise minimum codes for single messages that are known in advance.

3. We will use a method known as the Huffman Algorithm to produce our minimum code. This method uses a binary tree and gives us a chance to view a binary tree in a slightly different form. Instead of having each node point to its left and right child, each node will point to its parent and contain a

field, called childtype, which tells if it is a right child (RC) or a left child (LC). Our movement in the tree will, therefore, be from leaf to root rather than the other way around.

- The binary tree created is called a Huffman Tree and can be used to provide the minimum code desired. We will illustrate the method by deriving the minimum code for our motivational problem.

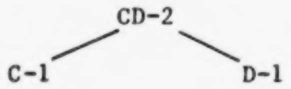
4. Our first step is to create a node for each of the symbols to be used. At this time, each node contains only a symbol and its frequency in the message, since they do not yet have parents. Our nodes are now: A-5, B-3, C-1, D-1 and E-2.

   - We will then go through these parentless nodes and choose the two nodes with the smallest frequency values. In our example, the nodes with C and D are the ones with the smallest frequencies.

     - In general, we will adopt the convention to use the symbols which occur first if there are more than two with these minimal values. We do this so that each of us will end up with the same minimum code.

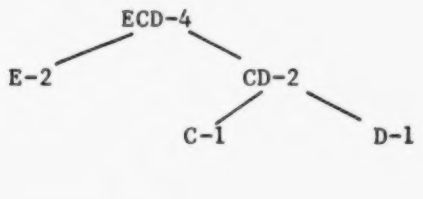
   - We now create a new parentless node whose left child is the node with the least frequency and whose right child is the node with the second least frequency. Again, since the values are the same, we choose the first occurrence (C) as the left child. This newly created node will have a symbol formed by the concatenation of the symbols from the left and right child, respectively. Its frequency is the sum of the frequencies of its two children. Our new node, in this case, is CD-2.

   - The parent fields of both C and D get set to point to this new node and their childtype fields get set to LC and RC, respectively. The current parentless nodes, in order, are A-5, B-3, E-2 and CD-2. A diagram of the binary tree formed by C, D and CD follows:

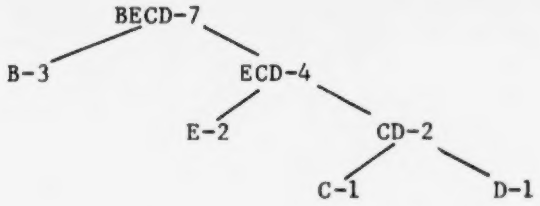


   - We again search through our list of parentless nodes and choose the two with the smallest frequency values. Finding them to be E and CD, respectively, we create the new node ECD-4 and set E to be its left child and CD to be its right child.

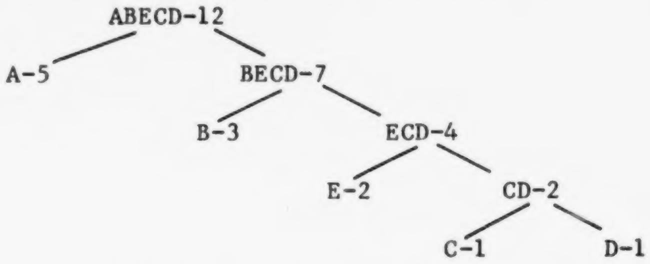
   - The remaining nodes without parents, listed in order, are: A-5, B-3 and ECD-4. The binary tree, being created from leaf to root, so far looks like this:



- In a similar manner, we next create the node BECD-7 leaving us with the parentless nodes A-5 and BECD-7, and the following tree:



- A final application leaves us with the single parentless node of ABECD-12 which is the root of the following Huffman Tree:



5. When the number of parentless nodes is reduced to just one, our Hoffman Tree is complete. The node that remains is the root of that tree.

   - To find the code produced for a symbol, you start at the leaf which contains that symbol and travel upwards to the root of the tree. At each step you obtain one bit of the code. If you travel to a parent from a left child, you add a bit value of 0 to the front of the bits you have collected thus far. If you are coming from a right child, you add a bit value of 1 to the front. Continue to add each bit value to the code until you arrive at the root.

     - In the tree above, the code for A is '0' since you arrive at the root after traveling to the parent of the node (leaf) A, which is a left child.

     - To obtain the code for B, begin at the leaf with value B. Your first step is to its parent BECD. Since B is a left child you start with the code value of '0'. Next, proceed to the parent of BECD, which is the root ABECD. Because BECD is a right child, add a '1' to the front of the code thus far. This value of '10' is the code for B.

     - The codes for C, D and E are obtained in a like manner and they are '1110', '1111' and '110', respectively.

6. By modifying the node structure slightly, the Huffman Tree can also be used to decode messages such as '1001100'. To do so, we must be able to travel from the tree's root to its leaves. Let us add two pointers to each node, called left and right, which point to the left and right child of the node.

- To decode a message, start at the tree's root. As each bit is read in, go to the left child of the node you are at if the value is 0 and to the right child if it is 1. Whenever you arrive at a node that is a leaf, output the value of that leaf. Then, return to the root and continue the process until all code bits have been processed.

- To decode the message '1001100', start at the root and go to its right child BECD for the first code bit of 1. For the next bit value of 0, go to the left child of BECD, which is the leaf B. At this point we output 'B' and go back to the root. The next bit value of 0 sends us to the leaf A so we output 'A'. The next three bits of the code will lead us to E and the final bit will again lead us to A. The message is thereby decoded into 'BAEA'.

7. Notice that every node in our Huffman Tree must either be a leaf or else it has both a left and right child. A tree with this property is called a strictly binary tree. You may recall that the binary trees used to represent algebraic expressions in Topic 83 also had this property.

    - The formula by which we can determine the total number of nodes for a strictly binary tree with N leafs is 2*N-1. Since the leafs of a Huffman Tree are the symbols we are coding, we can find the minimum code for N symbols with a Huffman Tree containing 2*N-1 nodes. We will use this fact in the program to follow.

8. Before we write a program to implement the Huffman Algorithm, let us discuss some of the data structures that we will be using. Recall that the nodes will contain a symbol, a frequency, a pointer to its parent, and a childtype (LC or RC).

    - We will use a variable of TYPE char for the symbol. We will choose to use dynamic nodes for our program rather than an array implementation. We will also declare an array of pointers to point to the nodes that are used in our tree. With these pointers we will be able to directly access any node in the tree.

    - Our program will contain three procedures. The Initialize (N) procedure will create the Nth node and input values for its symbol and frequency. The procedure Twosmall (P, Q, N) will search through the first N nodes and set P and Q to point to the one with the smallest frequency and the one with the second smallest frequency, respectively. The procedure Print(N) will print out the Nth symbol and its code.

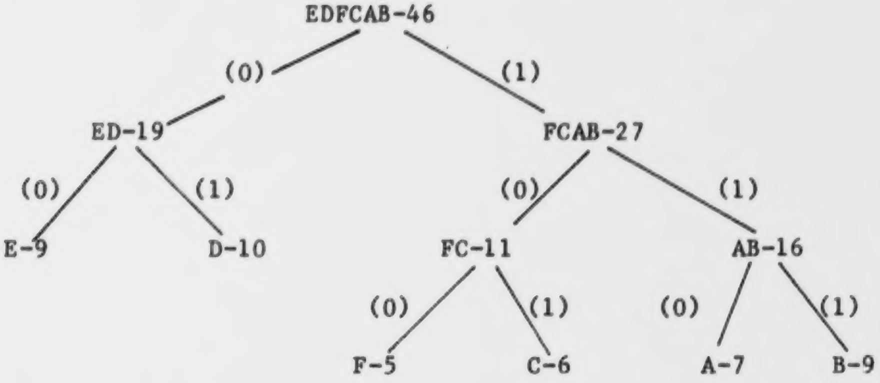
9. The following program finds the minimal code for given symbols and their frequency. The program follows the same procedure used previously to produce the code manually.

```
PROGRAM Huffman (input, output);
CONST
  maxnodes = 99; (* for up to 50 symbols *)
TYPE
  ctype = (LC, RC);
  nodeptr =↑nodetype;
  nodetype =
    RECORD
      symbol : char;
      freq : integer;
      parent : nodeptr;
      childtype : ctype
    END;
VAR
  node : ARRAY [1..maxnodes] OF nodeptr;
  numsymbols, I : integer;
  P, Q, R : nodeptr;

PROCEDURE Initialize (N : integer);  (* Creates the Nth node and input
          values for its symbol and frequency *)
VAR
  T : nodeptr;
BEGIN
  new (T);
  writeln ('Enter symbol number ',N,':');
  readln (T .symbol);
  writeln ('Enter its frequency: ');
  readln (T .freq);
  writeln;
  T .parent := NIL;
  node[N] := T
END; (* Initialize *)

PROCEDURE Print (N : integer); (* Prints the Nth symbol and its code *)
VAR
  T : nodeptr;
  code : ARRAY [1..100] OF char; (* Used to reverse order of output *)
  size, int : integer;
BEGIN
  size := 0;
  T := node[N];
  write (T↑.symbol, '-');
  REPEAT
    size := size + 1;
    IF T↑.childtype = LC THEN
      code[size] := '0'
    ELSE
      code[size] := '1';
    T := T↑.parent;
  UNTIL T↑.parent = NIL;
  FOR int := size DOWNTO 1 DO
    write (code[int]);
  writeln;
  writeln
END; (* Print *)
```

```
PROCEDURE Twosmall (VAR P, Q : nodeptr; N : integer);  (* Searches the
          first N nodes and returns the parentless nodes with the two
          smallest values.  Pointer P points to the smallest value and
          Q to the second smallest. *)
VAR
  S1, S2, int : integer;
BEGIN
  P := NIL;
  Q := NIL;
  S1 := maxint;
  S2 := maxint;
  FOR int := 1 TO N DO
    IF node[int]↑.parent = NIL THEN
      IF node[int]↑.freq < S1 THEN
        BEGIN
          S2 := S1;
            S1 := node[int]↑.freq;
            Q := P;
            P := node[int]
        END
      ELSE
        IF node[int]↑.freq < S2 THEN
          BEGIN
            S2 := node[int]↑.freq;
            Q := node[int]
          END
END; (* Twosmall *)

BEGIN (* Main Program *)
  writeln ('Enter the number of symbols: ');
  readln (numsymbols);
  FOR I := 1 TO numsymbols DO
    initialize (I); (* repeatedly take two smallest nodes and create
                       parent *)
  FOR I := (numsymbols + 1) TO (2 * numsymbols - 1) DO
    BEGIN
      new (R);
      R↑.parent := NIL;
      twosmall (P, Q, I-1);
      P↑.parent := R;
      Q↑.parent := R;
      P↑.childtype := LC;
      Q↑.childtype := RC;
      R↑.symbol := ' ';
      R↑.freq := P↑.freq + Q↑.freq;
      node[I] := R
    END;
  writeln;
  writeln ('Symbols and their code: ');
  writeln;
  FOR I := 1 TO numsymbols DO
    print (I)
END.
```

Application:

1. Draw the Huffman Tree and state the codes for the following symbols which are listed with their frequencies: A-7, B-9, C-6, D-10, E-9 and F-5.


EDFCAB-46
(0)
(1)
ED-19
FCAB-27
(0)
(1)
(0)
(1)
E-9
D-10
FC-11
AB-16
(0)
(1)
(0)
(1)
F-5
C-6
A-7
B-9

- The codes for A to F, respectively, are: '110', '111', '101', '01', '00' and '100'.

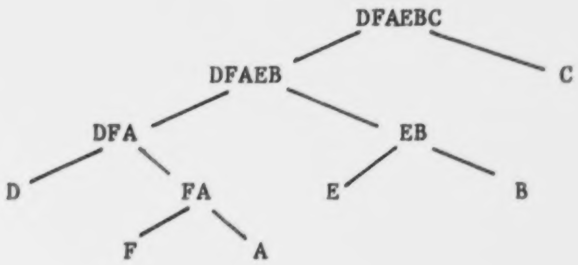
Summary:

1. What is a strictly binary tree?
2. What is a Huffman Tree?
3. How do we read the code for a symbol from a Huffman Tree?
4. How do we use a Huffman Tree to decode a message?
5. Briefly describe the Huffman Algorithm used to create a Huffman Tree.

Exercises:

1. Draw the Huffman Tree and state the code values for the following symbols which are given with their frequencies: A-15, B-10, C-31, D-12, E-30, F-62 and G-20.

2. Using the following Huffman Tree, encode the message "ACABFEDDB'. Note: We have not listed the frequencies on this tree.


DFAEBC
DFAEB
C
DFA
EB
D
FA
E
B
F
A

3. Using the above tree, decode the following:

   '000010110111001100000101'

4. Modify the program named Huffman so that the nodes include two more pointers, named left and right, which point to the left and right child of the node. Make all changes so that these pointers will be correctly set. Also, add the statement to the main program that sets a pointer named root to the root of the Huffman Tree.

5. Using the program and procedures as modified by exercise 4 above, write a procedure that reads in a coded message and outputs the decoded equivalent.

   - Note: This procedure will have to be called after the Huffman Tree has been created by the original main program.

TOPIC 87

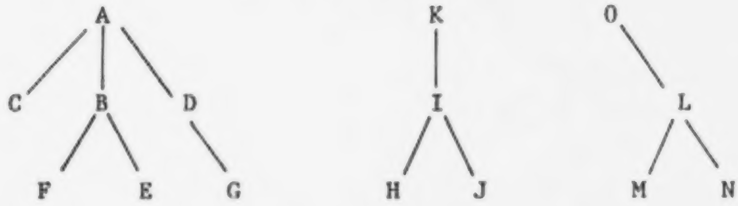
NON-BINARY TREES

Performance Objectives - Students will be able to:

- define a tree and related terms
- describe the array and dynamic variable implementations of general trees
- write a subprogram that adds a child to a node of a tree
- describe how a general tree can be represented using a binary tree

Vocabulary: tree, node, root, leaf, father, son, brother, parent, child, sibling, ancestor, descendant, level, degree, ordered tree, forest

Motivation: Tell why the following trees are or are not binary trees



- The third tree is the only one which is a binary tree. The first tree is not binary since A has three children. The second tree is also not binary, but the reason for this is not as obvious. Node K has only one child, but it is not clear whether I is a left child or right child of K. In a binary tree, each child must be designated as either a left or right child. Without such a designation, the tree is not binary because there are two different binary trees which could be represented by the diagram.

Aim: How to represent (non-binary) trees in Pascal.

Development:

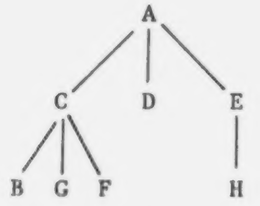
1. We define a tree to be a finite nonempty set of elements in which one element is called the root and the remaining elements are partitioned into m (m is a non-negative integer) disjoint subsets, each of which is itself a tree. This definition differs in three ways from our definition of a Binary Tree. First, an empty set is a Binary Tree, but it is not a Tree. Secondly, an element of a Tree may have more than two subtrees. Finally, the subtrees of a Binary Tree are designated as left and right subtrees while the subtrees of a Tree are an unordered set.

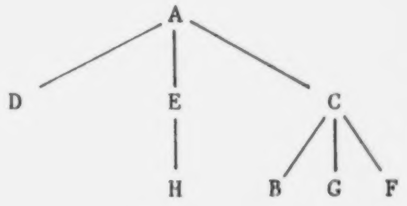
2. Most of the terms that we learned for binary trees are used with the same meaning for trees. These terms include: node, root, leaf, father, son, brother, parent, child, sibling, ancestor, descendant and level. One new term that is used with trees is the degree of a node which is defined to be the number of children that the node has. Therefore, a leaf has degree 0, while all other nodes have a positive degree.

3. A special type of tree is an ordered tree. In an ordered tree, the subtrees of a node are considered to be an ordered set. This is similar to the way we designated left and right subtrees for a binary tree.

   NOTE: This term has nothing to do with the ordered binary tree concept we previously introduced.

The following diagrams will illustrate an ordered tree and let us introduce some new terms.





These two trees would be equivalent as trees since the subtrees (and their subtrees in turn) have the same elements. But as ordered trees, these two trees are not the same. The ordering of the children of A is different for the two trees. We sometimes use the adjective oldest to refer to the first child of a node. The adjective youngest is similarly used to refer to the last child. In the first tree above, C is viewed as the oldest child of A and E is its youngest child.

A forest is defined to be an ordered set of ordered trees. The above diagrams could be viewed as forests containing the trees C, D and E. These two diagrams would represent two different forests since the trees are not arranged in the same order.

4. From now on, unless we specify otherwise, we will use the term tree to mean ordered tree.

5. It is important that you do not confuse the terms tree and binary tree. You may think that any nonempty binary tree is an ordered tree, if the left child is designated as being the oldest and the right child is designated as the youngest. This view may hold for strictly binary trees, but how do you show a node in the binary tree if it only has a right child. We cannot say it is the second oldest child if there is not a child who is older. It is, therefore, very important that you do not consider binary trees to be a special type of tree. They are two completely different entities even though they are defined in much the same way. This distinction will become even more important later on in the lesson.

6. Let us look at the nodes of a tree. These nodes must contain an information field and a collection of pointers to its children. If we knew the maximum number of children that a node would have, then we could use an array of pointers to point to the various children. But, since there is no limit to the number of children that a node may have, we must look for another solution. We will use a data structure that we studied previously, namely, a linear linked list. We will keep the children of a node in a linear linked list and use an additional pointer field, called next, in each node which will point to the node's next oldest sibling.

Thus, each node will contain three fields. One field is for information, another for the pointer to its next oldest sibling, and the third for a pointer to its oldest child.

7. The following declarations can be used to implement such a structure in Pascal with an array:

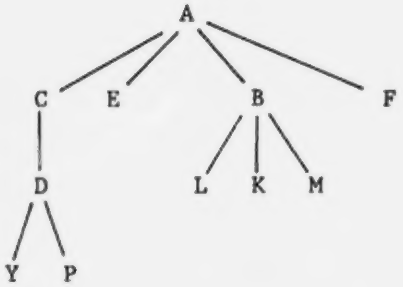
```
CONST
  maxnodes = 100;
TYPE
  nodeptr = 0..maxnodes;
  nodetype = RECORD
                 info : integer;
                 child, next : nodeptr
               END;
VAR
  node : ARRAY [1..maxnodes] OF nodetype;
  root : nodeptr;
```

8. To illustrate the representation, we take the data from the following array (and pointer variable root) and draw the corresponding tree.

NODE

root = 1

	info	child	next
1	A	3	0
2	B	8	6
3	C	4	5
4	D	11	0
5	E	0	2
6	F	0	0
7	K	0	9
8	L	0	7
9	M	0	0
10	P	0	0
11	Y	0	10



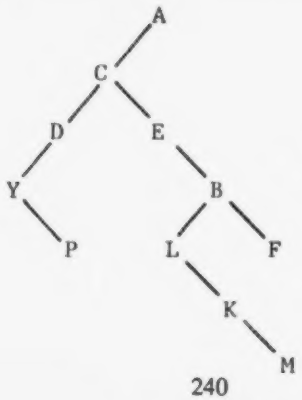
9. A similar implementation can be used with pointer variables. This dynamic representation can be declared as follows:

```
TYPE
  nodeptr = ↑nodetype;
  nodetype = RECORD
               info : integer;
               child, next : nodeptr
             END;
VAR
  root : nodeptr;
```

10. We now write procedures to handle the various tree operations. As an example, procedure addchild will add a child with value X to the node pointed to by pointer P; this child will be added as the youngest child of P. The procedure is written for the dynamic implementation. With a few changes the procedure can be modified to work with the array implementation, but this is left as an exercise.

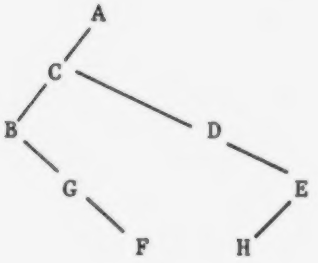
```
PROCEDURE Addchild (P : nodeptr; X : integer);
(* This procedure will add a child with value X as the youngest
   child of node P.  Assume P is not NIL *)
VAR
  Q, R : nodeptr;
BEGIN
  new (Q);
  Q↑.info := X;
  Q↑.child := NIL;
  Q↑.next := NIL;
  IF P↑.child = NIL
     THEN P↑ .child := Q
     ELSE
       BEGIN
         R := P↑.child;
         WHILE R↑.next <> NIL DO
           R := R↑.next;
         R↑.next := Q
       END
END;  (* Addchild *)
```

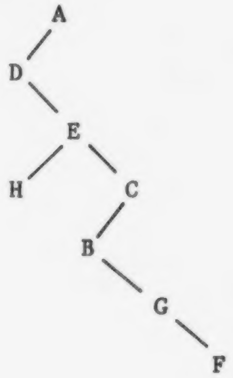
11. In the above implementations our nodes contain two pointers called child and next. If we view child and next as pointing to the left and right subtrees, respectively, of the node, then we can use this representation to obtain a binary tree. Let us draw the binary tree that is represented by the data in the array shown in paragraph 8.



12. We use this binary tree to represent the general tree that is shown in paragraph 8. Note that the above diagram can be obtained from the diagram of paragraph 8, by rotating the general tree diagram 45 degrees clockwise and changing some of the links. The parent-to-child links have been removed, except to the oldest child, and new links have been created between a node and its next oldest sibling. This representation of a general tree as a binary tree will be used in the next lesson to produce some interesting results.

13. As further examples of this binary tree representation of general trees, we include the binary trees that correspond to the two trees of paragraph 3.





Summary:

1. What is a tree?

2. What is meant by the following tree terms: leaf, parent (father), child (son), sibling (brother), ancestor, descendant, level, degree, ordered tree and forest?

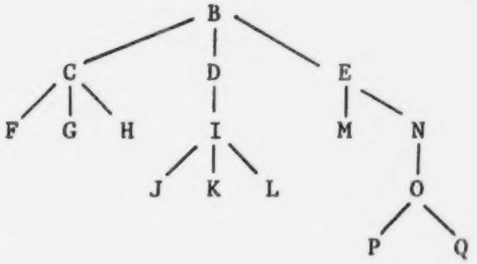
3. How do we represent trees in Pascal?

Exercises:

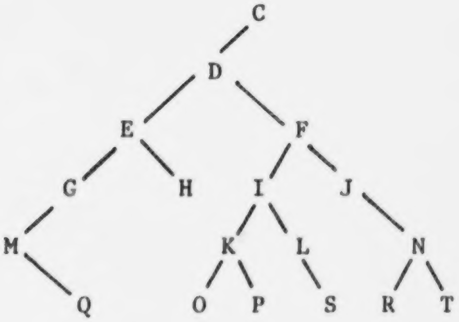
1. Rewrite the addchild procedure of paragraph 10 for the array implementation.

2. A slightly different tree node structure is sometimes used where the pointer child points to the youngest child and the pointer next points to its next youngest sibling. Rewrite the addchild procedure, using pointer variables, to add a youngest child to this type of node structure.

3. Given the following tree, draw its binary tree equivalent.


B
C
D
E
F
G
H
I
M
N
J
K
L
O
P
Q

4. Given the following binary tree representation of a general tree, draw the tree that it represents.


C
D
E
F
G
H
I
J
M
K
L
N
Q
O
P
S
R
T

TOPIC 88

TREE REPRESENTATION OF GENERAL EXPRESSIONS

Performance Objectives - Students will be able to:

- demonstrate how an ordered tree may be used to represent a general expression
- describe an algorithm that evaluates expressions represented by trees
- traverse a tree/forest in preorder, inorder and postorder

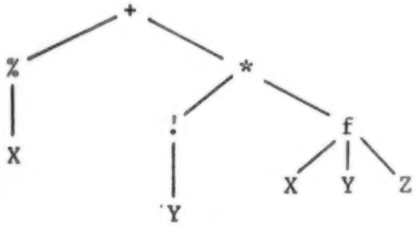
Vocabulary: traverse, preorder, inorder, postorder, prefix, infix, postfix

Motivation:

- In lesson 83 we represented algebraic expressions involving binary operators, by using a binary tree. Represent the following expression, which contains some non-binary operators, using a general tree:

- X + Y! * f(X, Y, Z)

Solution:



Aim: How to evaluate general expressions using trees.

Development:

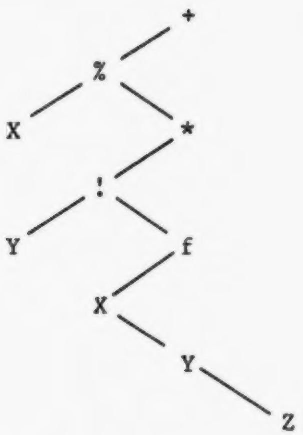
1. A tree allows us to represent any expression, not just those with binary operators. Note that in the above tree we used the symbol '%' for the unary minus operation. This was done to distinguish the minus symbol from the symbol for the binary operation of subtraction. Again, the leafs represent the operands while the other nodes represent the operations.

2. Recall that in lesson 83, the preorder, inorder and postorder traversals of the binary tree representation of an algebraic expression produced the prefix, infix (without parentheses) and postfix forms of that expression, respectively.

   In this lesson, we will define traversals for trees and forests and examine the various traversals for our trees which represent general algebraic expressions.

3. We use the binary tree representation of a tree or forest to define the traversal of the tree or forest; that is, to traverse a tree (forest) using the preorder, inorder and postorder methods, you traverse the equivalent binary tree using the preorder, inorder and postorder methods, respectively. The traversals of the binary tree will be done as described in lesson 83.

4. We will illustrate the various traversals for the tree from our motivational problem. First, we draw the equivalent binary tree for that tree:



5. Traversing this binary tree using the preorder method produces the following expression: +%X*!YfXYZ. This expression is the prefix form of our expression. In general, the preorder traversal of a tree which contains an expression will produce the prefix form of that expression.

6. Next, we traverse the binary tree, which represents our original tree, using the inorder method. This produces the following expression: X%Y!XYZf*+. This expression is the postfix (not infix) form of our expression. In general, the inorder traversal of a tree which contains an expression will produce the postfix form of that expression.

7. Finally, we traverse the binary tree representation of our tree using the postorder method. This produces the following expression: XYZYXf!*%+. This is not one of the usual forms of our expression. For this reason, the postorder traversal of trees is not used often. Furthermore, note that there is no easy way to get the infix form of our expression from the tree.

8. In order to create a tree for general expressions we will have to make our nodes capable of containing either an operator or an operand. We set up the nodes as a variant record. A node with an operation will have a field of type char for the operator symbol, a field for the pointer to its oldest child and a field for the pointer to its next oldest sibling. A node with an operand will have a field of type real for its value and a field for the pointer to its next oldest sibling. Since an operand was a leaf in the general tree, it will never have a child in the binary tree representation. The following declaration can be used for our nodes:

```
TYPE
  nodeinfo = (operator, operand);
  nodeptr =↑nodetype;
  nodetype =
    RECORD
      next : nodeptr;
      CASE tag : nodeinfo OF
        operator :  ( opcode : char;
                      child : nodeptr );
        operand  :  ( value : real)
    END;
```

9. We now write a function, called evaluate, which takes on the value of a tree which represents an expression; that is, evaluate(P) will take on the value of the tree whose root is pointed to by P. The function operates recursively as follows:

> Evaluate(P) returns the following, depending on the type of node that P represents. If P contains an operand, then evaluate(P) takes on its value. If P contains an operator, then the function evaluate(P) takes on the result obtained when the opcode operation is performed on its operand(s), where the evaluate function is again applied to each of these operands.

10. The following program implements the evaluate function. In order to use this function we must first create a binary tree to represent our expression. The procedure create(P) allows us to build a tree recursively. Each call to create will create a node; if the node is an operator it will also create its operand node(s). Remember that this procedure is recursive, so be careful to enter the nodes of the tree in their proper order. In addition to these procedures, another recursive procedure, called inorder, has been added to print out the expression in postorder.

<u>**NOTE**</u>: In the program below the % is the unary minus operation, the r is the unary reciprocal operation, the m operation returns the maximum value of its three operands, and the a operation returns the average of its three operands.

```
PROGRAM Tree;
(* This program will create, evaluate and print out a tree which
   represents a general expression *)

TYPE
  nodeinfo = (operator, operand);
  nodeptr =↑nodetype;
  nodetype =
    RECORD
      next   :↑nodeptr;
      CASE tag : nodeinfo OF
        operator : ( opcode  : char;
                     child   : nodeptr );
        operand  : ( value   : real)
    END;
```

```
VAR
  root : nodeptr;

PROCEDURE Create (VAR p : nodeptr);
(* This procedure will create a node pointed to by P *)
VAR
  ch : char;
BEGIN
  new (P);
  P↑.next := NIL;
  REPEAT
    write ('Enter O)perator or R)eal operand : ');
    readln (ch)
  UNTIL (ch = 'O') or (ch = 'R');
  IF ch = 'R'
    THEN
      BEGIN
        P↑ .tag := operand;
        write (' Enter Real Value : ');
        readln (P↑ .value)
      END
    ELSE
      BEGIN
        P↑ .tag := operator;
        write (' Enter Opcode : ');
        readln (P↑.opcode);
        CASE P| .opcode OF
          '%', 'r' : create (P↑.child);
          '+','-', '*', '/' : BEGIN
                                create (P↑ .child);
                                create (P↑ .child↑ .next)
                              END;
          'm', 'a'  : BEGIN
                        create (P↑ .child);
                        create (P↑ .child ↑ .next);
                        create (P↑ .child ↑ .next↑ .next)
                      END
        END
      END
END; (* Create *)

  FUNCTION Evaluate (P : nodeptr) : real;
  (* This function will return the value of the tree with root P *)

  VAR
    r1, r2, r3 : real;

  BEGIN
    IF P↑ .tag = operand THEN
      evaluate := P↑ .value
    ELSE IF P↑.opcode IN ['%','r','+','-','*','/'] THEN
      CASE P↑ .opcode DO
        '%': evaluate := -1* evaluate (P↑ .child);
        'r': evaluate :=  1/ evaluate (P↑ .child);
        '+': evaluate := evaluate (P↑.child) + evaluate (P↑.child↑.next);
      '-': evaluate := evaluate (P↑.child) - evaluate (P↑.child .next);
      '*': evaluate := evaluate (P↑.child) * evaluate (P↑.child .next);
      '/': evaluate := evaluate (P↑.child) / evaluate (P↑.child .next)
    END
  ELSE;
    BEGIN
      rl := evaluate (P↑.child);
      r2 := evaluate (P↑.child↑.next);
      r3 := evaluate (P↑.child↑.next↑.next);
      IF p↑.opcode = 'a' THEN
        evaluate := (rl + r2 + r3) /3
      ELSE IF (rl > = r2) AND (rl > = r3) THEN
        evaluate := rl
      ELSE IF r2 > = r3 THEN
        evaluate := r2
      ELSE
        evaluate := r3
    END
END; (* Evaluate *)

PROCEDURE Inorder (P : nodeptr);
(* The procedure will print out the tree using the inorder method;
   this will give a postfix expression.  Assume P is not NIL *)

BEGIN
  If P↑.tag = operand
    THEN
      BEGIN
        writeln (P↑.value : 10:2);
        IF P↑.next <> NIL
          THEN inorder (P↑.next)
      END
    ELSE
      BEGIN
        inorder (P↑.child);  (* cannot be NIL *)
        writeln (P↑.opcode);
        IF P↑.next <> NIL
          THEN inorder (P↑.next)
      END
END; (* Inorder *)

BEGIN  (* Main Program *)
  create(root);
  writeln;
  writeln ('Value is', evaluate(root):10:2);
  writeln;
  writeln ('Printing tree inorder gives the following');
  writeln ('which is in Postfix Form :');
  inorder (root)
END. (* Tree *)
```

Summary:

1. How do we traverse a general tree in preorder, inorder and postorder?

2. How do we obtain the prefix and postfix forms of an expression that is represented by a tree?

3. Briefly describe an algorithm to evaluate expressions represented by trees?

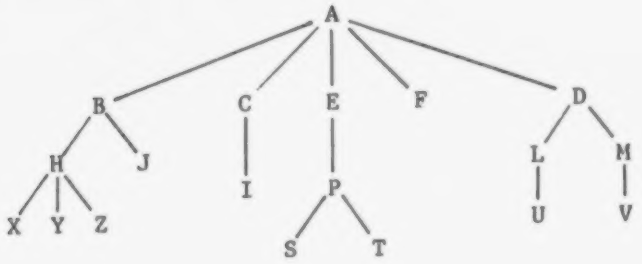
Exercises:

1. Represent the following expression in a tree and then draw the binary tree equivalent of the general tree: f(X,A*B-C,G!,-E)-D/(F+I)*(-P)

2. Use the binary tree created for exercise 1 to write the prefix and postfix forms of the expression.

3. Write a procedure for the tree program that will print out the prefix form of the expression represented by the tree.

4. Show the results of printing out the following tree using the preorder, inorder and postorder methods.



TOPIC 89

INTRODUCTION TO SORTING

Performance Objectives - Students will be able to:

- define terminology related to sorting
- discuss efficiency considerations that help determine the most appropriate sort in simple cases
- define Big-O notation
- compare the difference in efficiency between O(N) and O(N-squared) sorting time

Vocabulary: file, record, key, stable sort, internal sort, external sort, sorting by address, Big-O notation, quadratic sorting time

Motivation

- Given an array of 25 words (each word being a packed array of char), write a routine that will see if a given word is in the array.

```
TYPE
  wordtype = PACKED ARRAY [1..10] OF char;
  sentype = ARRAY [1..25] OF wordtype;

FUNCTION Isthere (VAR list : sentype; word : wordtype): boolean;
VAR
  index : integer;
  found : boolean;
BEGIN
  found := FALSE;
  index := 0;
  WHILE (index < 25) AND (NOT found) DO
    BEGIN
      index := index + 1;
      found := list[index] = word
    END;
  Isthere := found
END; (* Isthere *)
```

- How long this function will take to complete its job is dependent on the size of the array and where in the array, if at all, the desired word is located.
  - The minimum is one iteration; the maximum is 25 iterations; and the average is 13 iterations.
  - In more general terms, for N elements: the minimum is one iteration; the maximum is N iterations (for an array of size N); and the average is (N+1)/2 iterations.
- How can we make our search more efficient (make it more likely to run in minimal time)?

- We can sort our list. If this were done, the time necessary to determine that a word was not on the list would be decreased. It would also allow us to use a more efficient search algorithm, such as the binary search that was discussed in Topic 43.

Aim: What is a sort and how can we measure the efficiency of a sort?

Development:

1. Basic Definitions

   . File of size N: a collection of N pieces of information (used differently than when we discussed external files)

   . Record: one of the elements of information in a file (used differently than when we discussed a record variable). A record, in this case, may be a single piece of information of one data type.

     - Each record in a file can have several pieces of information. For example, consider the telephone book. Each entry contains the person's name (last, first), address and telephone number.

     - In this case the file is the telephone book and each entry would be considered a record.

     - When the file is sorted, we order it based on the name field (last, first-alphabetically). The field by which we sort is known as the key field.

2. We can consider a file to be an array of records. A key, K[I], must be associated with each record, R[I], of the file.

   . The key is usually, but not always, a field of the record.

   . Choosing the key is a very important part of the sort. If the phone book were sorted by address, or even by phone number, it would be useless in easing a search for a given person's address or phone number.

   . An internal sort is one in which all the records being sorted are stored in main memory.

   . An external sort is one in which some of the records being sorted are in a secondary memory or in an auxillary storage device. This may occur if all of the data to be sorted cannot fit into main memory at one time.

   . A stable sort is one in which the order of records with equal keys is not changed after a run of the sort is completed (that is, if K[I] = K[J] and R[I] precedes R[J], then when the file is sorted R[I] will still precede R[J]).

     - Stable sorts are desirable so that a file can be sorted by various keys in succession to produce a sort within a sort.

3. We can sort an actual file of records stored in an array, linked list, or other data structure. We can also sort by address, by sorting an array of pointers that point to records.

4. The time a sort takes depends on the size of the file.

   - We cannot talk about the time of a sort without knowing the machine it will be run on. As this factor is not under the control of the programmer, we talk about the time in units. These units are discussed independently of the machine and vary from machine to machine. But if two algorithms run in the same number of units, they will run in the same time on the same machine.

   - The table below indicates the time units necessary for a given routine to process files of various sizes.

file size	time units
10	20
20	40
50	100
100	200

   - Notice that a file of size N will be processed in 2N time units. We say that the time is of order N, or O(N). In general, if the processing time units is any constant multiple of N, the time is of order N.

5. Suppose study of a sorting algorithm shows its time to be expressable by the function $F(N)= .2N^2 + 3N$. Examine the following table:

N	$.2N^2$	3N	$.2N^2 + 3N$
10	20	30	50
50	500	150	650
100	2,000	300	2,300
1,000	200,000	3,000	203,000
10,000	20,000,000	30,000	20,030,000

   - As N becomes large, the significance of 3N decreases in comparison to $.2N^2$. For this reason we consider an algorithm with this function to be of order N-squared, or O(N-squared). This is referred to as quadratic sorting time.

Applications:

1. The time of a sort is $F(N) = (N + .1) * (N)$. What is its order?

   - Since $(N + .1) * N = N^2 + .1N$, the order is N-squared, or O(N-squared)

2. The time of a sorting algorithm is $F(N) = (N^5 - 3N^2 + 4) / (N + 1)$. What is its order?

- As N increases, the 4 and 1 become virtually irrelevant and F(N) approximates to $(N^5 - 3N^2) / N = N^4 - 3N$. The order is, therefore O(N-fourth).

- The order of an algorithm is the highest power of N in these cases..

- Which is a faster algorithm for sorting, one of O(N) or of O(N-squared)? Obviously an algorithm of O(N) would run faster.

3. Which algorithm would you choose to use in your program, one of O(N) or one of O(N-squared)?

   - In order to answer this question many things other than speed of the algorithm would have to be taken into consideration. We must consider: time to code and test the algorithm, amount of memory used by the algorithm, the quantity and type of data to be sorted, and the current state of the data.

Summary:

1. What factors make up the efficiency of a sort? (programming time, cost, speed of the algorithm, use of the sort)

2. What is a file? A record? A key field?

3. What is an internal sort? An external sort? A stable sort? A sort-by-address?

4. What is the Big-O Notation?

5. What is meant by quadratic sorting time?

Exercises:

1. Determine the order of a sort if the speed is given by $(N^2 - 4N + 5) / (.2N)$

2. Is the bubble sort routine a stable sort?

3. Make a chart showing the number of comparisons necessary for a bubble sort to sort files of size: 2, 4, 8, 16, 32, 64.

   - If each comparison takes one time unit, can you guess the order of the sort? (Assume that there is no boolean in the bubble sort routine to prematurely terminate the sort.)

TOPIC 90

SORTING - THE BUBBLE SORT

Performance Objectives - Students will be able to:

- discuss the efficiency considerations that help determine the most appropriate sort for a particular application
- explain why the bubble sort is of order N-squared
- show the state of the array items after each pass of the bubble sort

Vocabulary: Exchange sort, time/space trade off, primary key, secondary key, best case, average case, worst case, bubble sort, efficiency

Motivation:

- Suppose the Post Office has hired you to sort a given list of 100 names and addresses for easy mail distribution. The people are from different states in the United States.

- What factors would you have to consider in order to write an appropriate sorting algoritm?

Aim: How can we accurately assess and improve the efficiency of a sort?

Development:

1. Considerations for our Post Office sort:

   - What key should be chosen?

     - People are from different locations. The most appropriate key would be the zip code since mailing zones are determined by zip code. This would be our primary key, since more than one person could have a given zip code. We would choose the street address as a secondary key and then, possibly, the name as a third key. Remember, a key should be chosen so that we can later find a specific record based on the key.

     - The primary key is the main key by which we sort. This is used to divide the records into several smaller files. If the records are not in the desired order after being sorted by a primary key, a secondary key would be used. We would further divide the subfiles using the secondary key.

     - If further sorting is needed after using the secondary key, we could use a tertiary (third) key, and so forth.

     - Choosing the appropriate key, or keys, is a vital part of a sort.

- How many records are there in our file?
  - If our file has very few records, we would want the sorting algorithm that would be easiest to program. Since we are sorting a minimum number of records, the sorting time would be small whether our sorting algorithm was O(N) or O(N-squared).

- How often will the file require updating?
  - Each update would require that another sort be done. If there is adequate machine space, and time, and the sort routine will be run infrequently, it would not make sense to spend long periods of time programming the most efficient run-time sort.
  - A programmer may be able to justify the use of an inefficient sort in a specific situation. She must not, however, take the easy way out in all cases. Choosing an easily programmable but inefficient sort algorithm, in most cases, is not satisfactory.

- How will the sort be used at run-time?
  - Often a fast algorithm will use a lot of memory or carry with it some high overhead. There is often a time/space tradeoff. A programmer may have to optimize time (using a very fast algorithm if the usage warrants it) at the expense of space, or optimize space (limiting the amount of memory used) at the expense of time.
  - The decision as to which to optimize would depend on the particular algorithm as well as the type of machine. There is always a finite amount of memory your program can use. You must not exceed the maximum memory available, but there are times that you may be required to use substantially less than this maximum amount.
  - However, if your algorithm is for real-time use, the data required from the routine must be retrieved instantaneously. For example, in an air traffic program output received two seconds after input is useless and old. In an implementation like this, space would be sacrificed for time.

2. Order of a Sort

- So far we have discussed sorts of O(N) and $O(N^2)$.
  - To ensure that a file of size N is properly sorted, we must compare each item with some other item at least once. This implies that the minimum order of a sort would be O(N). However, an optimum algorithm of order O(N) does not exist.

Some Common Orders (In Increasing Order)

N
N * LOG N (close to O(N))
N^2
N^3
N!

- Our goal is to get as close to O(N) as we can.

3. Review of the bubble sort

- The bubble sort is an example of an exchange sort, a sort where elements of the file are compared and, if necessary, exchanged with one another.

- Consider our bubble sort routine:

```
PROCEDURE Bubblesort (VAR list: numarray; size: integer);
  (* list is an array of reals,  size is the number of array items
  used *)
VAR
  aswitch: boolean;
  I,
  index: integer;
BEGIN
  index := size (* initializes index *)
  REPEAT
    aswitch := FALSE;
    index := index-1;
    FOR I := 1 TO index
      IF list[I] > list[I+1] THEN
        BEGIN
          switch(list[I], list[I+1]); (* switch is previously defined
                                         to swap 2 real values *)
          aswitch := TRUE
        END
  UNTIL NOT aswitch
END; (* Bubblesort *)
```

4. What is the order of a bubble sort?

- In the best case, the case requiring the fewest comparisons, the loop would run through once. This would give us a sort of order O(N).

- In the worst case, the case requiring the most comparisons, the loop would run through (N-1) + (N-2) + (N-3) + ... + 2 + 1 times. This gives us a sort of order $O(N^2)$.

- In the average case, we would get an $O(N^2)$.

- What would be the initial state of the array that would yield a best case situation?

    - If the array were already ordered, we would have a best case situation as no items would be switched and aswitch would remain false.

  - What would be the initial state of the array that would yield a worst case situation?

    - If the array were in reverse order or if the smallest item is in the last location, we would have a worst case situation.

5. See Topic 42 for a trace of the bubblesort procedure.

Summary:

1. What is the best case, worst case, and average order for a bubble sort?
2. What is meant by the term time/space tradeoff?
3. What is a primary key? secondary key?
4. What is an exchange sort?

Exercises:

1. A file consists of records, each of which contains the first name, last name, address, and social security number for a person. Write a bubble sort routine to sort the file by social security number.
2. Follow the array items listed below as they are sorted by a bubble sort routine:

   fish, dog, zebra, cat, hippo, hog, calf, snake

TOPIC 91

BINARY TREE SORT

Performance Objectives - Students will be able to:

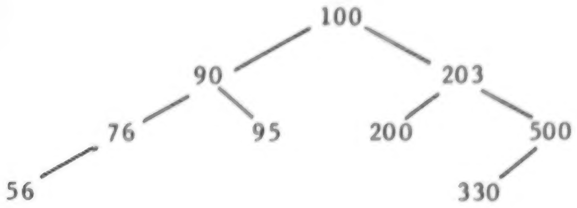
- write a binary tree sort algorithm
- discuss why the relative efficiency of the binary tree sort depends on the original order of the data as well as the amount of data
- show that the efficiency of the binary tree sort is O(N * LOG N)
- demonstrate how a list of items is sorted using the binary tree sort

Vocabulary: binary tree, binary tree sort, ordered binary tree, binary sort (search) tree

Motivation:

- What is an ordered binary tree sort?
  - An ordered binary tree is a binary tree where all elements stored in the left subtree of a node X are less than the element stored at X, and all elements stored in the right subtree of X are greater than or equal to the element stored at X. An ordered binary tree is sometimes referred to as a binary sort (search) tree.
- Place the elements of set S in an ordered binary tree.

S = (100, 90, 76, 203, 56, 95, 500, 330, 200)



Aim: How can we write an efficient binary tree sort?

Development:

1. An inorder traversal of an ordered binary tree would produce elements in sorted order.
   - How would we write the code to perform this sort?
     - Possible Solution

```
PROGRAM Binsort (input, output);
TYPE
  ptr = ↑node;
  node =
    RECORD
```

```
          info : integer;
          left,
          right : ptr
        END;
    VAR
      root : ptr;
      I : integer;
    BEGIN
      root := NIL;
      WHILE NOT eof DO
        BEGIN
          readln (I);
          Insert (root, I)  (* Adds a node with value I to tree 'root' *)
        END;
      Inorder (root)  (* Prints out elements of tree 'root' in order *)
    END. (* Binsort *)
```

. What would procedure Insert look like?

```
    PROCEDURE Insert (VAR rt: ptr; X : integer);
    BEGIN
      IF rt = NIL THEN (* empty subtree *)
        BEGIN
          new (rt);
          rt↑.info:= X;
          rt↑.left:= NIL;
          rt↑.right:= NIL
        END
      ELSE
        IF (X< rt↑.info) THEN (* goes in the left subtree *)
          Insert(rt↑.left, X):
        ELSE (* goes in the right subtree *)
          Insert(rt↑.right. X);
    END; (* Insert *)
```

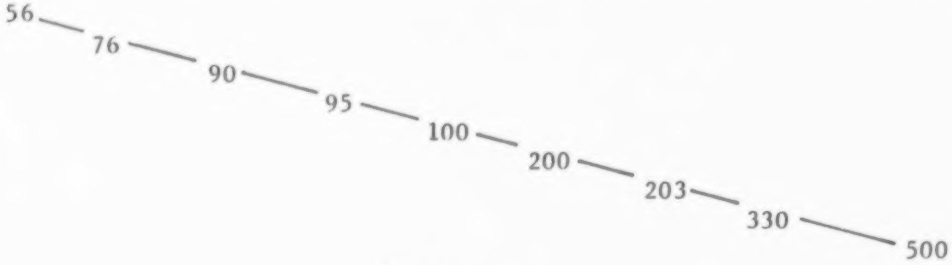
. Procedure Insert is a recursive algorithm to insert items in a binary search tree. Procedure Inorder is left as an exercise.

2. Consider the following data of sets S and S':

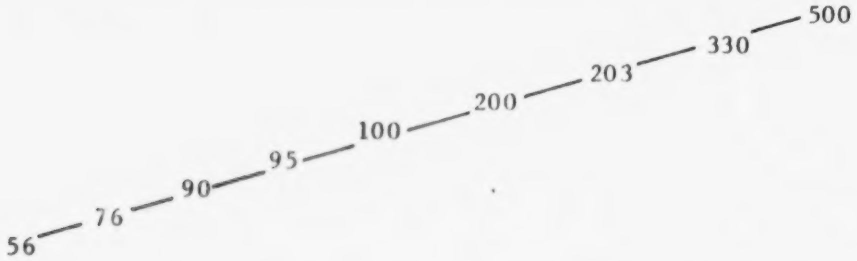
S = (100, 90, 76, 203, 56, 95, 500, 330, 200)

S' = (56, 76, 90, 95, 100, 200, 203, 330, 500)

. Note that S = S'. What would the ordered tree created by S' look like?



- Now consider the set: S'' = (500, 330, 203, 200, 100, 95, 90, 76, 56) What would its tree look like? (Note that S = S' = S'')



- S' and S'' will output the same ordered data as the tree created by S if inorder traversal is used. However, the trees created by S' and S'' are linear and of level 8 (generally for a file of N records the level would be N - 1).

- A sort using S' or S'' would take more time than one using S since the increased number of levels requires more comparisons. The efficiency of the binary tree sort depends on the original order of the data as well as the quantity of data.

3. What is the efficiency of a binary tree sort?

- We already saw that the relative efficiency of the sort is dependent on the original data, its size and order.

- A pre-sorted set of data would result in a tree of only left (or right) subtrees. If we study this worst case (either the tree created by set S' or S'') we see:

  - The first node is inserted (one comparison).

  - The second node is inserted (two comparisons).

  - The third node is inserted (three comparisons).

  - This continues, and at the Nth node there would be N comparisons.

- The number of comparisons therefore is equal to: $1 + 2 + 3 + 4 + 5 \ldots + (N-1) + N = N(N+1) / 2 = N^2 / 2 + N / 2$ or O(N-squared).

  - This, again is the worst case of the binary tree sort.

- In the best case, the depth of the tree is equal to $[\mathrm{LOG}_2(N+1)-1]$

  - In this situation, all levels, except possibly the bottom, are completely filled, producing a complete binary tree. In a complete binary tree of level K, the number of nodes is equal to $2^{K+1}-1$ (in our case this represents the ceiling of our best case); therefore $N = 2^{K+1}-1$ and $\mathrm{Log}_2(N+1) = K+1$, which makes $K = \mathrm{Log}_2(N+1)-1$.

- The number of nodes at any level of the tree, level J for example, would be 2^J. The number of comparisons necessary to place a node at level J, where J is not equal to 0, would be J+1. The total number of comparisons is therefore approximately equal to:

$$\sum_{L=1}^{D} 2^L(L+1)$$

where D is the level of the full tree. This expression is of order O(N * LOG(N)), as will be demonstrated:

$$\sum_{L=1}^{D} 2^L (L + 1) = 2(2) + 2^2(3)+2^3(4) + \ldots + 2^D(D+1)$$

$$<= (2 + 2^2 + 2^3 + \ldots + 2^D)(D+1) = (2^{D+1}-2)(D+1)$$

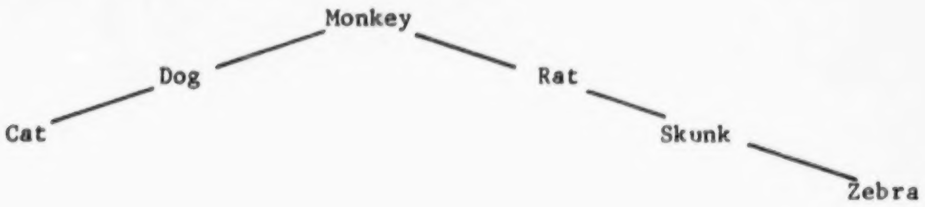
$$= 2^{LOG_2(N+1)}-2)(LOG_2(N+1)) = (N+1-2)LOG_2(N+1) = O(N*LOG(N))$$

- In the case of the binary tree sort, the order of the best case is close to the order of the average case. Thus, the binary tree sort has order O(N*LOGN). In the worst case, the first element has one comparison, the second two, the third three, and so on. The total number of comparisons is 1+2+3...+N, which is of order $O(N^2)$.

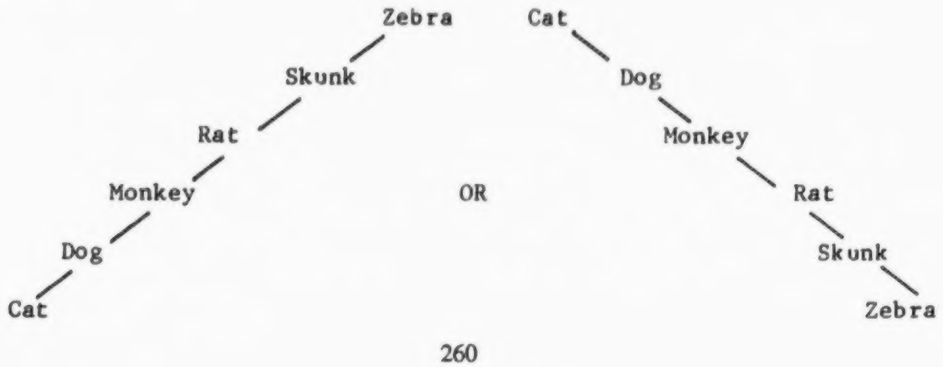
Applications:

1. Draw a binary search tree for the following set of words.

S = (monkey, rat, dog, cat, skunk, zebra)

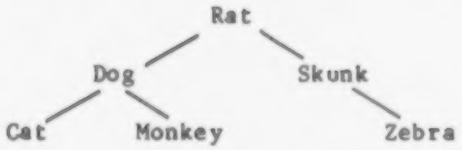


- What could the worst case binary tree for the above set look like?

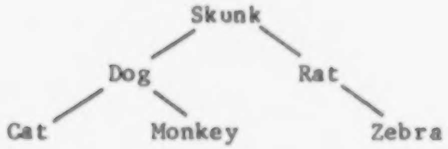


. What could the best case binary tree for the above set look like?

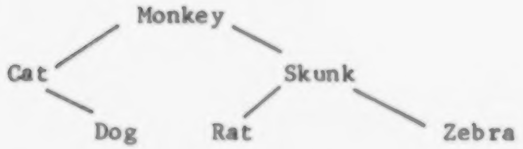
Possible Solution:


Rat
Dog
Skunk
Cat
Monkey
Zebra

or


Skunk
Dog
Rat
Cat
Monkey
Zebra

or


Monkey
Cat
Skunk
Dog
Rat
Zebra

. We see the best and worst cases are not unique.

Summary:

1. What are the rules for inserting elements in a binary sort tree? (All elements stored in the left subtree of a node have key values less than the value stored in the node, and all elements-stored in the right subtree of a node have key values greater than or equal to the value stored in the node).

2. What determines the relative efficiency of a binary search tree?

3. What is the average efficiency of a binary tree sort? [O(N*LOG N)]

Exercises:

1. What order of the elements of set S above would produce the best case? The worst case?

2. Insert the letters h, c, j, a, f, g, i, b into a binary search tree.

TOPIC 92

SELECTION SORT

Performance Objectives - Students will be able to:

- define a selection sort
- write a routine for a selection sort
- prove that a selection sort is O(N-squared)
- show the state of array items after each pass of the selection sort

Vocabulary: selection sort

Motivation:

- How would you sort a set of real numbers read from an external file?
- To read and sort reals we must first know if we will read all of the data first and then sort, or if we wish to sort the data as it is read in. This would be a matter of minimizing overhead without threatening the application.
  - Input/Output is expensive and it might be advantageous to read all of the data before doing any processing. This method would perhaps suggest a bubble sort.
  - On the other hand, we may wish to minimize memory allocation or require the data to be constantly ordered for our particular application. These considerations indicate that a binary tree sort may be more applicable.
  - Are there other sorts we can use?

Aim: What is a selection sort?

Development:

1. A selection sort is a sort where successive elements of the data are selected from the file and placed in their proper position. For example, given the list of numbers -- 3, 6, 5, 1, 10, 4 -- we can place the elements in order, starting with the smallest, as follows:

3 6 5 1 10 4	Find the smallest value contained in positions 1 through 6 and interchange it (if necessary) with the value in position 1.

1 6 5 3 10 4	Position 1 now contains the smallest value. Next, find the smallest value in positions 2 through 6 and interchange it with the value in position 2 so that position 2 contains the second smallest value.
1 3 5 6 10 4	In a similar manner, switch the third smallest value into position 3 after examining positions 3 through 6.
1 3 4 6 10 5	Similarly, switch the fourth smallest value into position 4.
1 3 4 5 10 6	Finally, switch the fifth smallest value into position 5.
1 3 4 5 6 10	The array is now sorted.

2. What is the order of a selection sort?

   . First, we do N-1 comparisons, then N-2 comparisons, then N-3 comparisons, and so forth, until at the last time we do 1 comparison. The number of comparisons is therefore: $(N-1)+(N-2)+(N-3)+ \ldots +2+1 = N(N-1)/2$ or $(N^2-N)/2$. This function indicates that a selection sort is O(N-squared).

3. How would we code a selection sort?

```
FOR I := 1 TO N - 1 DO
  FOR J := I + 1 TO N DO
    IF info[I] > info[J] THEN
      swap(info[I], info[J]);
```

4. Can we improve the efficiency?

   - Swap is a procedure that is called each time data has to be changed. To improve efficiency, it would be better to have an index for the location of the current smallest item and then call swap once (if needed) at the end of the iteration. This technique was used in the manual example that we previously examined.

```
FOR I := 1 TO N - 1 DO
  BEGIN
    index := I;
    FOR J := I + 1 TO N DO
      IF info[J] < info[index] THEN
        index := J;
      IF index <> I THEN
        swap(info[I], info[index]
  END; (* FOR I *)
```

Application:

1. Consider the following code:

```
PROCEDURE Whatdo (VAR info: arraytype; N : integer);
VAR
  large : datatype;
  I,
  J,
  index : integer;
                         (* datatype is a globally defined type and
                         arraytype is globally defined to be an array
                         of datatype*)
BEGIN
  FOR I := N DOWNTO 2 DO
    BEGIN
      large := info[I];
      index := I;
      FOR J := 1 to I - 1 DO
        IF info[J] > large THEN
          BEGIN
            large := info[J];
            index := J
          END;
      X[index] := X[I];
      X[I] := large
    END
END;
```

   - What does procedure Whatdo, do?

   - This is another version of the selection sort. This version uses "large" as a temporary holder of data as opposed to calling procedure swap. The other main difference is that this procedure puts the largest item in the last position, the second largest in the next to last position, and so forth.

- We may avoid unnecessary actions by testing if index <> I before setting X[index] := X[I] and X[I] := large. If index = I, these assignments are unnecessary. Notice, however, that this adds a boolean test at each iteration as well.

2. Given the array of integers: 36, 12, 19, 17, 20, 63, 15 and using our procedure whatdo, how will the array look at each iteration?

```
n=7
i=7         large=15      index=7
j=1 to 6
36                                       36
12          large=36      index=1        12
19                                       19
17          large=63      index=6        17
20          Interchange X[6] with X[7]   20
63            (i.e., 63 with 15)         15
15                                       63

i=6         large=15      index=6
j=1 to 5
36                                       15
12          large=36      index=1        12
19                                       19
17                                       17
20          Interchange X[1] with X[6]   20
15                                       36
63                                       63

i=5         large=20      index=5
j=1 to 4
15                                       15
12                                       12
19                                       19
17          No interchange is necessary  17
20                                       20
36                                       36
63                                       63

i=4         large=17      index=4
j=1 to 3
15                                       15
12                                       12
19          large=19      index=3        17
17                                       19
20          Interchange X[3] with X[4]   20
36                                       36
63                                       63
```

```
i=3          large=17       index=3
j=1 to 2
15                                        15
12                                        12
17                                        17
19           No interchange is necessary  19
20                                        20
36                                        36
63                                        63

i=2          large=12       index=2
J=1 to 1
15                                        12
12                                        15
17           large=15       index=2       17
19                                        19
20           Interchange X[1] with X[2]   20
36                                        36
63                                        63
```

Summary:

1. What is a selection sort?

   . A selection sort chooses successive elements of a file and places them in their proper position in the sorted file.

2. How can we make sort algorithms more efficient?

   - eliminate unnecessary instructions
   - avoid excessive procedure calls
   - avoid unnecessary boolean tests
   - avoid redundant statements
   - minimize overhead

Exercises:

1. Which sort is more efficient, a selection sort or a bubble sort? Give reasons for your answer including a discussion of the order of each sort.

2. A Quadratic Selection Sort is performed the following way: Divide the n elements of the file into k groups where k=ROUND(SQRT(k)). Find the largest element of each group and insert it into an auxiliary array. Find the largest element in this array and that element will be the largest element in the file. Place that element into its proper place and then replace it in the auxiliary array with the next largest element of its group. Continue this process until the file is sorted.

   - What is the order of the quadratic selection sort?
   - Compare its efficiency with the selection sort and the bubble sort.

TOPIC 93

INSERTION SORT

Performance Objectives - Students will be able to:

- define an insertion sort
- write an insertion sort algorithm
- prove that the insertion sort is O(N-squared).
- discuss how a simple insertion sort can be improved using a binary insertion or list insertion
- demonsrate a shell sort algortihm.

Vocabulary: insertion sort, binary insertion sort, list insertion sort, shell sort, diminishing increment sort

Motivation: What is a common method used to sort a set of test papers by grade?

- First, pick up one of the test papers. Next, pick up a second paper and insert it in a position so that the two papers are ordered. The third paper is then inserted into a position so that the first three papers are ordered. If each succeeding paper is inserted in a similar fashion, then the papers will be sorted when the last paper is inserted. This method of sorting is an example of an insertion sort.

Aim: What is an insertion sort?

Development:

1. An insertion sort is a sort that inserts elements into an existing ordered file, in such a manner that the resulting file remains ordered.

   - Procedure insort implements an insertion sort.

```
PROCEDURE Insort (VAR nums: arraytype; n: integer);
(* arraytype is a global type *)
VAR
  k, i, y: integer;
  found: boolean;
BEGIN
  FOR K := 2 TO N DO
    BEGIN (* FOR k *)
      y := nums[k];
      i := k-1;
      found := FALSE;
      WHILE (i >= 1) AND (NOT found) DO
        IF y < nums[i] THEN
```

```
          BEGIN (* if *)
            nums[i+1] := nums[i];
            i := i-1
          END (* if *)
        ELSE
          found := TRUE;
      nums[i+1] := y
    END (* FOR k *)
  END; (* whatdo *)
```

2. Execute PROCEDURE Insort on the following array:

```
nums [1]---   3
nums [2]---  50
nums [3]---   7
nums [4]---   8
nums [5]---  -3
nums [6]---   5
```

```
nums
3
50      n=6
7
8
-3
5

k=2   (* first iteration *)
y=50    i=1     found=false

3     y< nums[1] is false         3
50      found := true             50
7     nums[2] := y                7
8                                 8
-3                                -3
5                                 5

k=3   (* second iteration *)
y=7     i=2   found=false

3     y<nums[2] is true           3
(50)7   nums[3]:=nums[2]          7
(7)50   i := 1                    50
8     y< nums[1] is false         8
-3      found := true             -3
5     nums[2] := y                5

k=4   (* third iteration *)
y=8     i=3     found=false

3     y< nums[3] is true          3
7       nums[4] := nums[3]        7
(50)8   i := 2                    8
(8)50 y< nums[2] is false         50
-3      found := true             -3
5     nums[3] := y                5
```

```
k=5    (* fourth iteration *)
y=-3       i := 4 found=false

(3)-3 y<nums[4] is true          -3
(7)3    nums[5] := nums[4]        3
(8)7    i := 3                    7
(50)8 y<nums[3] is true           8
(-3)50    nums[4] := nums[3]     50
5       i := 2                    5
      y<nums[2] is true
        nums[3] := nums[2]
        i := 1
      y<nums[i] is true
        nums[2] := nums[1]
        i := 0
        nums[1] := y

k=6    (* fifth iteration *)
y=5     i := 5    found=false
-3    y<nums[5] is true          -3
3       nums[6] := nums[5]        3
(7)5    i := 4                    5
(8)7  y<nums[4] is true           7
(50)8   nums[5] := nums[4]        8
(5)50   i := 3                   50
      y<nums[3] is true
        nums[4] := nums[3]
        i := 2
      y<nums[2] is false
        found := false
      nums [3] := y
```

- At the kth iteration of procedure insort the first k elements are sorted and the k+1 element is being inserted into its proper position in the sorted array. For example when k=2 the first 2 elements are sorted and the third is being inserted so that at the next iteration (the third) the first three elements are sorted and the fourth is being inserted, etc.

3. What is the efficiency of an insertion sort?

In the best case, the case when the file is already sorted (unlike the binary tree sort when the already sorted file gives us the worst case), there is only one comparison made at each iteration giving us an algorithm of order O(N).

In the worst case, the case when the file comes sorted in reverse order, there are (N-1) comparisons at the first iteration, (N-2) comparisons at the second iteration, etc. This gives us:
(N-1)+(N-2)+(N-3)+.......+3+2+1=(N-1)(N)/2 comparisons, an algorithm of order O(N-squared).

4. How can we improve on the efficiency of this algorithm?

   a. We can use a binary search rather than a linear search in order to find the proper position of the k+1 element in the sorted file of k terms. As this may decrease the number of comparisons needed to find the correct location of insertion, it would appear to increase the efficiency of our algorithm. However, we still must change the position of the other array items (push them down into their proper location) once the location is found. This makes the *binary insertion sort* O(N-squared) as well.

   b. We can sort by address using a *list insertion sort*. Our original array would be an array of pointers, one for each of the original records. In this case, a pointer can be adjusted in less time than it takes to move the record which it references. This reduces the time to perform an insertion even though the time required to search for the proper point of insertion is the same. This algorithm requires extra memory as an array of pointers would be set up. If your records are large, the extra memory used is compensated for by the savings in insertion time.

5. The *shell sort* is an improvement on the insertion sort, with a better efficiency rating.

   The shell sort is based on the notion that the best case for an insertion sort occurs when the file is nearly sorted, coupled with the idea that a relatively small amount of data, N, may be efficiently sorted by an (N-squared) algorithm. What we are doing with a shell sort is sorting smaller subfiles to get the large file closer to being in order before it is sorted as a whole.

   The shell sort (also known as the *diminishing increment sort*) sorts several subfiles of the original file. These subfiles contain every kth element of the original file, where k is determined by the programmer as the increment.

   1. For example, given a file of 30 records and k=5, 5 subfiles would be established as follows:

1.	x[1]	x[6]	x[11]	x[16]	x[21]	x[26]
2.	x[2]	x[7]	x[12]	x[17]	x[22]	x[27]
3.	x[3]	x[8]	x[13]	x[18]	x[23]	x[28]
4.	x[4]	x[9]	x[14]	x[19]	x[24]	x[29]
5.	x[5]	x[10]	x[15]	x[20]	x[25]	x[30]

   Each of these subfiles is sorted (using an insertion sort). The file is then considered as a whole and a new smaller k is chosen. The process is then repeated. This goes on until k=1 and we are sorting the entire file.

EXAMPLE:

```
n=8
GIVEN:              36      58      42      39      -2      90      87      40
X[..]:              1       2       3       4       5       6       7       8
--------------------------------------------------------------------------------

                                         40   42
K=5:                36   90 |  58   87 | 42   40 |  39 | -2
X[  ]:              1    6  |  2    7  | 3    8  |  4  |  5

ARRAY:              36   58    40   39   -2   90   87   42
X[  ]:              1    2     3    4    5    6    7    8
--------------------------------------------------------------------------------

                                    | -2   42   58 |
k=3:                36   39    87   |  58  -2   42 |  40   90
X[  ]:              1    4      7   |  2    5    8 |  3    6

ARRAY:              36   -2    40   39   42   90   87   58
X[  ]:              1     2    3    4    5    6    7    8
--------------------------------------------------------------------------------

                                                    58   90
K=2:                36   40    42   87 | -2   39    90   58
X[  ]:              1    3     5    7  |  2   4     6    8

ARRAY:              36   -2    40   39   42   58    87   90
                    1     2    3    4    5    6     7    8
--------------------------------------------------------------------------------

                    -2   36    39   40
k=1:                36   -2    40   39   42   58    87   90
X[  ]               1     2    3    4    5    6     7    8

COMPLETED ARRAY:    -2   36    39   40   42   58    87   90
X[  ]:               1   2     3    4    5    6     7    8
```

The sorted file at k=1 is the entire file, and our sort is complete.

The sequence of the increments is very important. The increments must be relatively prime; if not, the same subfiles would be sorted repeatedly. For a good choice of increments, the efficiency of a shell sort is $O(N * (LOG\ N)^2)$.

SUMMARY

1. What is an insertion sort?

2. What is the order of an insertion sort?

3. How can the efficiency of an insertion sort be improved by using a binary insertion or a list insertion?

EXERCISES

1. Write a binary insertion sort routine.

2. Write a list insertion sort routine.

3. Given the following file of reals,

32.1 -34.4 -54.3 34.1 43.6 76.5 43.1 32.1 -54.3 -32.1

   a. What values would you choose as your increment values for a shell sort? Give a reason for your choice?

   b. Show what would happen to this file if it were to be read into an array and sorted with our procedure shell and the increment values chosen in a.

   c. Is the shell sort a stable sort (give a reason for your answer)?

TOPIC 94

MERGE SORT

PERFORMANCE OBJECTIVES - Students will able to:

- Show how two ordered arrays can be merged.
- Explain the code for a merge sort procedure.
- Write a merge sort procedure.
- Discuss the efficiency of a merge sort.

MOTIVATION:

- Given two sorted arrays, a and b, each of size n, write a procedure that will create a third sorted array, c, of size 2n which contains all elements of a and b.

```
PROCEDURE  Mergethem  (a, b: firstarray; n: integer; VAR c: secondarray);

VAR
  apt, bpt, cpt:  integer;  (* control index of each array*)
BEGIN
   apt:=1;
   bpt:=1;
   cpt:=1;
   WHILE (apt<= n) AND (bpt<= n) DO
        BEGIN (* while *)
             IF a[apt]< b[bpt] THEN
                  BEGIN (* if *)
                    c[cpt] := a[apt];
                    apt := apt+1
                  END
             ELSE
                  BEGIN
                    c[cpt] := b[bpt];
                    bpt := bpt +1
                  END;
             cpt := cpt+1
        END; (* while *)

(* fill in remaining elements *)
   WHILE apt<= n DO
        BEGIN
             c[cpt] := a[apt];
             cpt := cpt + 1;
             apt := apt + 1
        END;
   WHILE bpt <= n DO
```

```
        BEGIN
            c[cpt] := b[bpt];
            cpt := cpt + 1;
            bpt := bpt + 1
        END
END; (* Mergerthem*)
```

A procedure, similar to the one above, can be written to sort two files of different sizes. This procedure can be reapplied over and over to merge several small ordered files into one.

This method would be very useful if there was a file to be sorted that was too large to be sorted in memory. Parts of the file could be sorted and stored and then these subfiles could be merged to reform the original file in sorted order.

Aim: What is a merge sort?

Development:

1. A merge sort is a sort that divides a file into subfiles of size 1 and then combines them to recreate the original file in a sorted order. It proceeds as follows: Pairs of subfiles of size 1 are merged to create ordered subfiles of size 2. These subfiles ae then merged to create ordered subfiles of size 4. The process contines until we are left with 1 subfile which is the original file in ordered form.

2. A recursive algorithm that performs a merge sort is:

```
Mergesort:
    split file in half;
    IF size (lefthalf)>1
        THEN mergesort (lefthalf);
    IF size (righthalf)>1
        THEN mergesort (righthalf);
    Merge (lefthalf, righthalf);
```

3. Consider an array of range[low..high]. It is first divided into arrays of range[low..(low+high) DIV 2] and range[(low+high) DIV 2+1..high]. These subarrays are then continually sub-divided until we have created all subarrays of size 1. The following recursive algorithm divides the array, as previously described, and then merges the subarrays to reform the original array in sorted order.

```
PROCEDURE Mergesort (VAR arr:  arraytype; low, high:  integer);
VAR
   mid:  integer;
BEGIN
   IF low < high THEN (* continues until low>=  high *)
       BEGIN (* if *)
           mid  := (low + high) DIV 2;
           mergesort (arr, low, mid);
           mergesort (arr, (mid+1), high);
           merge(arr,low,mid,high)
       END; (* Mergesort *)
```

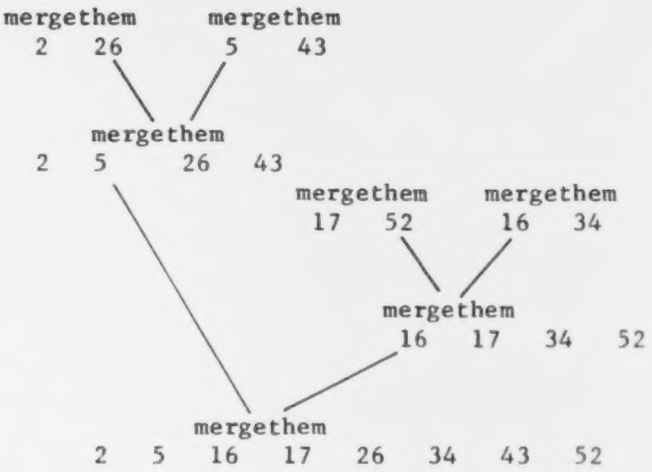
This code is simply a translation of our recrusive algorithm. Now we must write procedure merge. We see by our parameter list that we cannot use our earlier procedure mergethem. Our new merge could be written as follows:

```
PROCEDURE Merge (VAR arr:  arraytype; low, mid, high:  integer);
VAR
   temp:  arraytype; (* temporarily holds the sorted subfile *)
   i, left, right, tempcount:  integer;
BEGIN
   tempcount  := low;
   left  := low; (* starts the left subfile *)
   right  := mid+1; (* starts the right subfile *)
   WHILE (left<= mid) AND (right<= high) DO
     BEGIN (* while *)
       IF arr[left]<= arr[right] THEN
         BEGIN
           temp[tempcount] := arr[left];
           left  := left +1
         END
       ELSE
         BEGIN
           temp[tempcount] := arr[right];
           right := right +1
         END;
       tempcount := tempcount+1
     END; (* while *)
(* we now fill in the remaining items *)
   WHILE (left < = mid) DO
       BEGIN
         temp[tempcount] := arr[left];
         tempcount := tempcount+1;
         left := left +1;
       END; (* for *)

(* now restore the items sorted and temporarily stored in temp *)
   FOR i := low to right-1 DO
     arr[i] := temp[i]
END; (* Merge *)
```

EXAMPLE

arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]	arr[7]	arr[8]
26	2	43	5	17	52	16	34

```
  3     2     4   1     6     5     7        (order of recursive splitting)
26   2     43   5   17    52     16    34
```


mergethem
2 26
mergethem
5 43
mergethem
2 5 26 43
mergethem
17 52
mergethem
16 34
mergethem
16 17 34 52
mergethem
2 5 16 17 26 34 43 52

4. What is the efficiency of a merge sort?

   a. We divide the N records in half giving O(log N).

   b. We merge them in a linear algorithm O(N).

   c. This tells us that the merge sort is O(N * log N).

SUMMARY

1. What is a merge sort?

2. What is the efficiency of a merge sort?

3. What are the advantages of a merge sort?

EXERCISES

1. Write a non-recursive merge sort routine.

2. Would you prefer the recursive of non-recursive merge sort? Give reasons for your answer (be sure to discuss efficiency considerations).

TOPIC 95

SEARCHING OVERVIEW AND SEQUENTIAL SEARCH

Performance Objectives - Students will be able to:

- discuss the action of a searching algorithm
- discuss the relationship between searching and sorting
- code a sequential search of a file

Vocabulary: Searching, table, key, internal key, external key

Motivation:

- How would you go about finding last night's math homework in a messy pile of papers on your desk?

- The most natural approach is to begin at the top of the pile, examining each paper in turn, until you recognize last night's homework, at which point you stop.

  - In your search, you look for certain characteristics of the page, rather than reading its entirety. For example, you might look for the title ('Math Homework' or 'Annoying Algebra'). But if the pile contains many nights' math homework, you may need to look at the date as well, and maybe other characteristics which would help you to distinguish among them.

Aim: How can a table of records be searched?

Development:

1. The process of retrieving certain information from a larger collection of information is called searching. A computer-based searching algorithm often proceeds much as the day-to-day situation described above. We will develop a class of algorithms to model the situation.

2. To do this, we make certain assumptions about the environment. We assume the data is organized as records, which are further divided into fields.

3. The set of records may be organized in different ways. It may be stored in an array, a file, a linked list, or in some other data structure. Any such set of records is referred to as a table.

4. We further assume the data in each field can be compared. One field is called the key for the search. This field plays the same role as the key in a sorting algorithm. We wish to find a record or set of records for which the information in the key field matches some given data.

   - Keys that are fields of a record are called internal keys, while keys that are kept outside the record, but contain pointers to the records, are called external keys.

5. If a record is found in which the key matches the data, the search is said to terminate successfully; otherwise, it is said to terminate unsuccessfully. In either case, various actions can be performed.

   - If successful, the information in the record may be copied and used in a process external to the sorting algorithm. The record may be deleted from the table or replaced by a new record (updated). It may be removed from its current position in the table and reinserted in a new position.

   - If the search is unsuccessful, often a new record is inserted, one which contains information originally missing from the table.

   - In Pascal, different ways of storing a table are convenient for different actions. Deletion and insertion are easy in a table stored as a linked list, but rather cumbersome in a table stored in an array or file.

6. A simple searching algorithm, the exact analogue of the search described for the math homework, is called a <u>sequential</u> search.

   - The algorithm checks each record in turn to see if the key matches the given data.

     - If we are searching for only one occurrence of the key, the algorithm stops as soon as it encounters a match.

     - If we are searching for all occurrences of the key, the algorithm proceeds to the end of the table, checking each record.

7. The function "Seqsearch" searches a table composed of an array of records for a specified integer key value. If the key value is found in a record, the corresponding array index is returned; otherwise, a 0 is returned.

   - Suppose "table" has been declared as ARRAY [1..N] OF rectype; where rectype is a record which contains a field called "key."

```
FUNCTION Seqsearch  (VAR table : arraytype; keyvalue: integer): integer;
  VAR
    I : integer;
    found : boolean;
  BEGIN
    found := false;
    I := 1;
    WHILE (I<= N) AND (NOT found) DO
      IF table[I].key = keyvalue THEN
        BEGIN
          seqsearch := I;
          found := true
        END
      ELSE
        I := I + 1;
    IF NOT found THEN
      seqsearch := 0
  END;
```

8. There are several advantages in storing a table as a linked list. Among them are: the table's size can change dynamically, and insertions and deletions are simpler.

   - The function "Seqsearch2" searches a table, maintained as a linked list, for a given key value. If the key value is found in a node, the function returns a pointer to this node; otherwise, a NIL pointer is returned.

```
FUNCTION Seqsearch2 (table : listptr; keyvalue : integer) : listptr;
VAR
  I : integer;
  found : boolean;
  P : nodeptr;
BEGIN
  found := false;
  P := table;
  WHILE (P<>NIL) AND (NOT found) DO
    IF P↑.key = keyvalue THEN
      BEGIN
        seqsearch2 := P;
        found := true
      END
    ELSE
      P := P↑.next;
  IF NOT found THEN
    seqsearch2 := NIL
END;
```

Summary:

1. What is a sequential search?

2. What are the advantages in storing a table as a linked list?

Exercises:

1. A table of data is stored in a Pascal array of records with the following structure:

```
student =
  RECORD
    name : nametype;
    age : integer;
    sex : sextype;
    average : integer
  END;
```

. Write functions or procedures to accomplish the following tasks:

  - Given an identification number, return the record of the student with that identification number.
  - Given a student's name, return the record of the student with that name. If there is no such student, report this fact.

TOPIC 96

EFFICIENCY OF SEARCH ALGORITHMS

Performance Objectives - Students will be able to:

- tell how to measure the efficiency of a sequential search
- compare the efficiency of a sequential search with that of the binary search
- discuss techniques which can be used to improve the efficiency of a sequential search

Vocabulary: self-ordering table, move-to-front method, transposition method

Motivation:

- A "Beat The Clock" contestant has ten seconds to find a word in the dictionary. The dictionary has no "tabs", but is arranged in normal alphabetical order. The contestant can prepare the dictionary any way he wants before the ten seconds begins, but he has no way of knowing what the word will be. How should he prepare the dictionary?

- Aside from the obvious (within reach, right side up, etc.), students should be able to guess that the contestant would do best by opening the dictionary to some place near the middle. Then the distance from the actual location of the word to the place where one is looking is, on the average, smallest.

Aim: How can we measure the efficiency of searching algorithms?

Development:

1. The intuition behind the above situation is that, on the average, a word occurs in a sorted list of words in the middle. This intuition, in fact, applies to any table, sorted or unsorted. It shows that one can expect to go through about half the records in the table on searching sequentially.

   - If the table has N elements, the expected number of comparisons is N/2. Hence the efficiency of a strictly sequential search is O(N).

2. We have not assumed anything yet about the arrangement of the records in the table being searched. Certain arrangements may allow a given algorithm to run more efficiently.

   - In particular, a sorted table can be an asset. If the key for searching was also the key for sorting, then a sequential search algorithm can be terminated (successfully or unsuccessfully) as soon as the position of the object in the table has been passed. There are other advantages of a sorted table associated with other forms of searching algorithms.

   - However, a sorted table is not always best. It requires more overhead to produce and maintain. In some instances this overhead may be prohibitive.

   - In addition, it may happen that a table is maintained in sorted order with one key, but search and retrieval is done with a different key. Although some data structures allow the maintenance of both orders simultaneously (a multilinked list is an example), the overhead is typically even greater.

3. There are even cases when the structure of a sorted table can be a liability.

   - For instance, suppose we are searching for a single occurence of a given key, which is likely to occur often in the table. If the table is sorted on that key, all occurrences of the key are clumped together in the table. Hence we may expect, on the average, to have to search halfway down the table.

   - If, however, the table is not sorted, the multiple occurrences of the key are likely to be distributed more evenly throughout the table. The algorithm will terminate successfully, on the average, after having searched through a smaller part of the table.

4. If a table is to be maintained in sorted form, and searched sequentially, then there is a theoretically optimal order for storing the records. This can be done by placing the record which is most likely to be needed first, the next most used record second, the third most used record in third place, and so on.

   - That is, if P(I) is the probability that the Ith record is needed for a given search, we should arrange the table so that P(1) > P(2) > P(3) > ..., and, in general, so that if J > K, then P(J) > P(K). (We have not provided for records whose retrieval is equally probable, but it is easy to do so.)

     - For example, if the table contains records of student averages and it is likely that we will need information about the best students, then the records of students with the higher averages should be placed earlier in the table.

   - Unfortunately, the numbers P(I) are rarely known in advance, and may even vary for different occasions on which the table is searched. There are several techniques for ordering the table in these cases.

     - These methods are based on the assumption that if a record is retrieved, then it becomes more likely that it will be needed again. For instance, the most used books in a library are also the most likely books to be checked out.

   - We can use this assumption by reordering the table. If a record is retrieved, we can place it earlier in the table. But how much earlier? The move-to-front method places it first. This is making a rather large "bet" that the record will be used again. It can give poor results if a large number of records are accessed once, and then never again. But in another case, it can result in substantial savings.

- Another method is to place a more "moderate bet" on the record retrieved by advancing its position more slowly. The most common way of doing this is to exchange it with the record just before it in the table. Then the records used most often will slowly "creep" to the front of the table. This technique of exchanging records is called the transposition method.

5. The binary search algorithm (see Topic 43) is the most efficient way for searching a sorted sequential table of records. Since each comparison reduces the number of records to be examined by a factor of 2, the binary search time is approximately $O(\text{Log}_2 N)$. It is important to realize that the binary search can only be used with arrays.

Summary:

1. How can the action of a searching algorithm also be used to restructure the data in a table?
2. What are three techniques to improve efficiency in a sequential search?
3. What are the advantages and disadvantages of using a sorted table when performing a sequential search?
4. What is the time efficiency of a sequential search? binary search?

TOPIC 97

INDEXING A TABLE

Performance Objectives - Students will be able to:

- describe how an index can be used to help organize a table
- write algorithms to construct an index for a table
- write algorithms to use an indexed table in searching

Vocabulary: index, indexed sequential search, primary index, secondary index

Motivation:

- How do you look up an entry in an encyclopedia?

- The answer is, of course, that the entries are ordered alphabetically. But an encyclopedia generally has many volumes. To help you select the correct volume, the bindings of these volumes usually contain names of the first and last entries in them.

  - For example, in looking up the entry for Nicaragua, one might select the volume labelled "Napier to Oriskanny." These names coded on the bindings of the volumes of the encyclopedia constitute an index to the entries.

Aim: How to search an indexed sequential table.

Development:

1. Using an indexed table works in much the same manner. You search sequentially through the index until you find the entry which would come just before the key (note that this usually involves reading the entry which comes just after the key). The process is similar to inserting the key in the index as a table on its own.

2. The analogy can be taken further. Upon opening the selected volume, we find that each page is also indexed--the names of the first and last entries on each page are copied at the top of the page.

   - This is an example of a secondary index. Frequently, the first index of a table grows so long that it also must be indexed. An index to an index is called a secondary index. In some cases, a third or fourth index may be considered, although space and time considerations make further indexing impractical, and other means of organizing the search are used.

   - Note, however, that a sorted table with several levels of indexing begins to resemble a tree. Several very sophisticated ways of searching a table can be developed from this line of thought.

3. To create an index, the table must be sorted. Then the keys of certain records are copied into a new table, together with the location of the record in the original table. The new table is sorted sequentially and used to find the starting point for a sequential search of the original table.

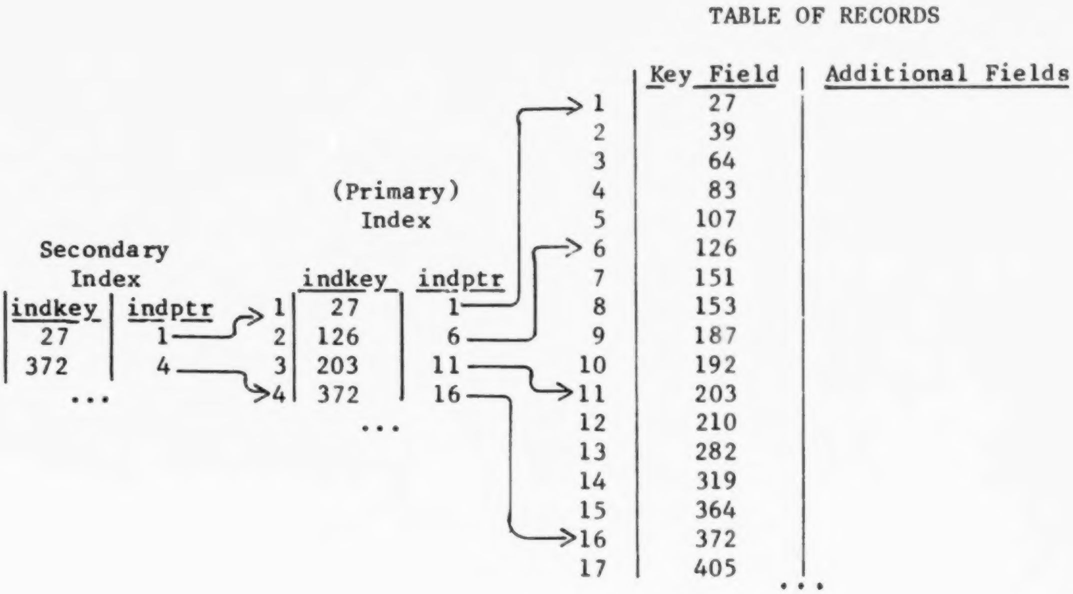


TABLE OF RECORDS
Key Field
Additional Fields
1 27
2 39
3 64
4 83
5 107
6 126
7 151
8 153
9 187
10 192
11 203
12 210
13 282
14 319
15 364
16 372
17 405
...
(Primary) Index
indkey indptr
1 27 1
2 126 6
3 203 11
4 372 16
...
Secondary Index
indkey indptr
27 1
372 4
...

- Suppose we wanted to search the table of records for a record whose key field contains a value of 282. We could compare 282 sequentially with each key in the (primary) index until we find a value that is greater than 282 (that is, 372) or reach the end of the index. Look at the index pointer in the prior position to the one just found in the index, and start searching the table of records at this location (that is, 11). Proceed to compare 282 with each key field until a match occurs or location 15 (one less than the index pointer 16, which corresponds to 372) has been reached. In our example, the search is successful and 282 is found in location 13 of the table.

4. A function that performs an indexed sequential search is written below. Assume that the arrays table and index have been previously declared as arraytype1 and arraytype2, and have respective sizes of tablesize and indexsize. When a key value is found, the position of the match in the table of records is returned; otherwise, a zero is returned.

```
FUNCTION Indexseq (VAR table : arraytype1; VAR index : arraytype2;
                   keyvalue : integer) : integer;
VAR
  I,
  pos,
  start,
  finish : integer;
```

```
    found : boolean;
  BEGIN
    I := 1;
    found := false;
    WHILE (I<= indexsize) AND (NOT found) DO
      IF index[I].indkey > keyvalue THEN
        found := true
      ELSE
        I := I + 1;
    IF I<>1 THEN
      start := index[I - 1].indptr
    ELSE
      start := index[1].indptr;
    IF I<= indexsize THEN
      finish := index[I].indptr - 1
    ELSE
      finish := tablesize;
    pos := start;
    found := false;
    WHILE (pos<= finish) AND (NOT found) DO
      IF table[pos].key = keyvalue THEN
        found := true
      ELSE
        pos := pos + 1;
    IF found THEN
      indexseq := pos
    ELSE
      indexseq := 0
  END;
```

5. Maintenance of an indexed table requires some care if insertions and deletions are to be performed on the original table.

   - When deleting a record from the base table, maintenance should be done on that record's index entry. Otherwise, a person searching only for the existence of the record may conclude from the presence of the index key that the record is present in the table as well. This danger can be avoided in several ways.

     - One way is to replace the index entry with another entry when a record it points to is deleted. This takes a little more time in running the algorithm.

     - Another method would be to maintain a marker along with each index entry which tells whether the index entry is actually present in the file. This method is quick, but requires more space in the index table.

   - Insertions to an indexed table can cause even worse trouble.

  - If they happen to occur in one small region of the original table, then the advantage of an index begins to be lost. The growing section of the table requires a longer and longer sequential search for retrieval. An extreme case might begin to look like the following:

INDEX	TABLE
apple	apple
	bagel
cat	cat
	diner
	dipper
	dog
	doodle
	doorknob
	duel
	dwindle
elephant	elephant
	frog

  - This table started life as a list of words, with every other word appearing in the index. The programmer anticipated that insertions would occur evenly throughout the table. But for some reason, many insertions occurred around the letter "D". As a result, a word beginning with the letter D is now more difficult to retrieve than other words. To find the word "dwindle", for instance, requires more comparisons than finding any other word in the file.

- For optimal efficiency, the entries to the index table should be evenly distributed among the entries to the main table. In the encyclopedia analogy, this is just saying that the ideal encyclopedia will have an equal number of entries in each volume, rather than having many thin volumes and a few fat ones.

6. An indexed table is a sorted table which is accessed by a series of sequential searches. First we search the index sequentially, then we search a section of the table sequentially. Its efficiency depends on the fact that we are shortening the sequential searches being performed. Thus it is not too surprising that searching an indexed table has an efficiency of O(N), although the coefficient of N in the expression can be considerably less than that for a strict sequential search.

7. Though an indexed search substantially reduces the amount of time to do a sequential search of key values in a table, an increase of allocated space is required. When there is a need for many insertions and deletions, a linked list table implementation improves efficiency. However, additional space is needed for pointers.

Summary:

1. How can we improve searching efficiency on a sorted table?

2. What is an indexed sequential search?

3. What are the advantages and disadvantages of using an indexed sequential search?

4. What is the efficiency of searching an indexed table?

Exercises:

1. Take an alphabetical list of students in your class, and select names which would form a good index to this table.

2. A sorted table is stored as a Pascal file of 30,000 records. The records have eight fields each, including the name of the person whose record it is. An index to this table is stored as another file of records. Each record has two fields: a name field and a field indicating the sequential position of the record in the main file which bears that name.

   . Write a program to search this table for a given name. What difficulties would you encounter if you had to code algorithms for insertion and deletion into a table with this structure?

3. Write a function that performs an indexed sequential search on a linked list.

TOPIC 98

SEARCHING A BINARY TREE

Performance Objectives - Students will be able to:

- demonstrate how to form a balanced binary search tree
- show how insertions and deletions can be made in a binary search tree
- discuss the efficiency of using binary search trees

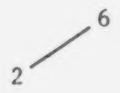
Vocabulary: binary search tree

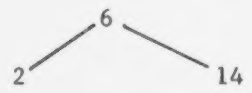
Motivation:

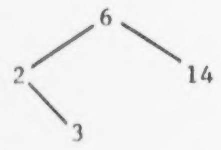
- Use a binary tree to sort the following list of numbers:

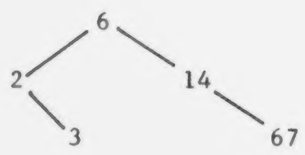
  6, 2, 14, 3, 67, 35, 24, 13, 34

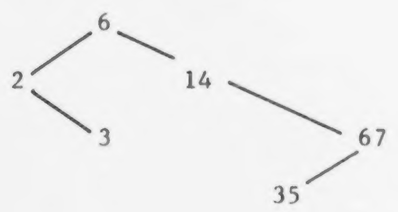
- This can be done by putting the first number given (6) in the root of the tree, then placing each successive number on a branch to the left or right, depending on whether it is smaller or larger than the node being visited:

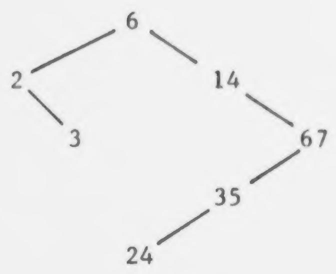


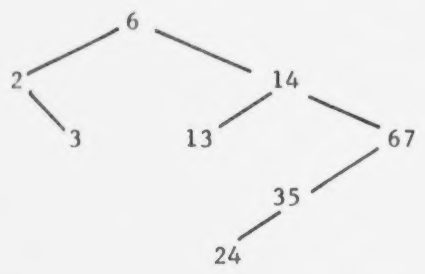


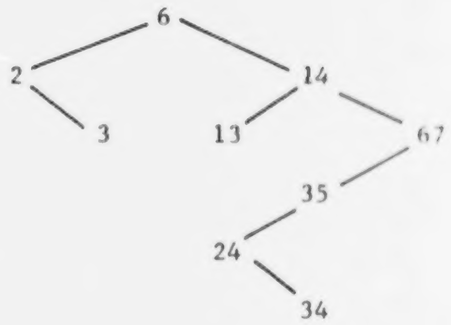












- If the tree is traversed in inorder, the original set of numbers will be recovered in order.

Aim: How can binary trees be used for searching?

Development:

1. The tree formed in the binary tree sort process contains within it an ordering of the table. And because of the branching, searching such a tree can be considerably shorter than a corresponding sequential search of a sorted table.

   - For instance, in the tree above, the number 67 can be found using three comparisons: Go to the right at the first level (because $67 > 6$), and to the right again at the second level (because $67 > 14$), then stop at the third level, because the key has been discovered. This is a substantial saving over the strict sequential search in which the number 67 would turn up only at the very end of the file.

2. For some entries in the above tree, the savings are not as dramatic. For instance, to search for 34 or 24, we perform almost as many comparisons as in a sequential search, because these numbers occur at a very deep level of the tree.

3. A tree is most efficient when it has few levels.

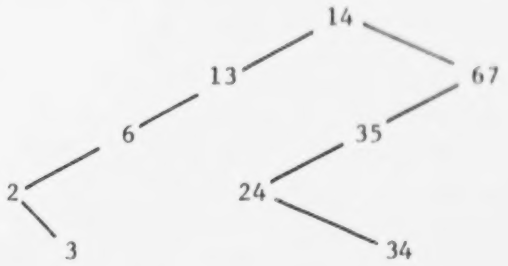
   - The above tree is not as efficient as it could be because of the long "tail" on the right side which reaches down many more levels than the other side.

   - Also, there is plenty of room on higher levels for new numbers. It would be better if the tree could be balanced--and still maintain the order of the file--so that it uses fewer levels to store the same numbers.

4. A little experimentation shows that the tree's imbalance was caused by selecting the number 6 as the root. This number is not very representative of the average entry in the tree, so most entries are on one side of it. The early occurrence of the number 67 also caused more imbalance.

- Some algorithms employ partial knowledge of the contents of the unsorted file to prevent the search tree from being unbalanced.

- There are also algorithms, called "rotations", to balance an unbalanced tree. In the example above, we could "pick up" the tree by the node with 14 in it, and shake it out, to give it more balance:



  - Rotation algorithms examine the nodes of a tree and balance it by "picking up and shaking out", while adjusting the nodes to preserve the binary structure.

- A sorted array can produce many binary search trees. When we take successive elements from array B, an unbalanced binary tree results (see figure 1). If we take the middle element as the root and look at the remaining elements recursively as left and right subtrees, we obtain the balanced binary tree shown in figure 2.

Array B

1	2	3	4	5	6	7	8	9	10	11
12	17	18	39	47	49	62	74	81	88	93

FIGURE 1

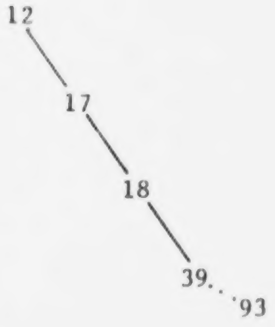
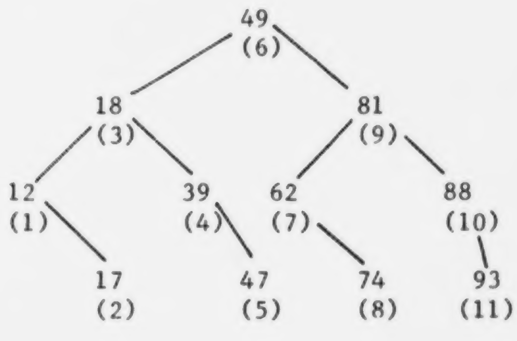


FIGURE 2

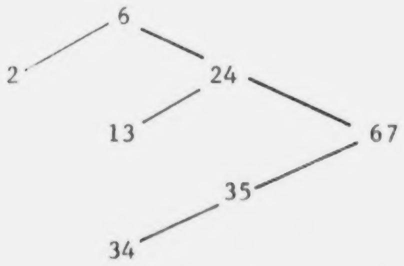


5. Insertions into a binary search tree are not difficult. There is always room for more nodes in the correct place. The only problem may be that insertions begin to make a tree look unbalanced.

- Deletion from a tree is more difficult. We can always delete a node which is a leaf. If a node has only one child, that child can take the place of the parent in the tree (this usually involves only a shift of pointers).

- But it is more difficult to delete a node with two children. If we want to preserve the order properties of the tree, we must replace this node with the node that comes after it in order--its inorder successor. This can be found by the usual inorder traversal algorithm, starting with the node to be replaced. We must replace the number to be deleted with its inorder successor, then delete the node containing the successor. It is not hard to see, from the action of an inorder traversal, that the inorder successor of a node with two children will not have two children. Thus deleting this node reduces to one of the easier cases described above.

- For example, consider the problem of deleting the node containing 14 from our original binary tree. We can find its inorder successor by traversing in inorder the subtree starting at 14. We find that the successor is 24. We copy the 24 into the node which originally contained 14. The original node containing 24 has no left child (otherwise this child would come between 14 and 24 in an inorder traversal), so we can delete it easily. This operation preserves the inorder of the tree, so we can search it just as we did before the deletion.



- An advantage of using a binary tree over an array is the relative ease of doing insertions and deletions.

6. In a well-balanced binary tree with K levels, we can store $2^{K}-1$ numbers. Hence a tree with N elements will have, on the average $\log_2 N$ levels. The number of levels is also the number of comparisons we must expect in searching a tree. Hence the efficiency of a binary tree search is O(log N).

Summary:

1. What is a binary search tree?
2. What type of binary search tree improves search efficiency?
3. How can we insert/delete nodes from a binary search tree?
4. What is the efficiency of a binary search tree?

Exercises:

1. A table of records is stored as a binary tree. The records contain information about the names and ages of various people, and the tree is ordered alphabetically by name. Write a program to traverse the tree and print out the names of all people of age 21 or younger.
2. A set of numbers is stored in a binary tree. Given another number, write an algorithm to determine if the number is already in the set. If it is, delete it from the tree; otherwise, insert it in the tree. Make sure your insertions and deletions preserve the inorder of the tree.

TOPIC 99

HASHING

Performance Objectives - Students will be able to:

- define a hash function and hash clashes (collisions)
- describe how collisions can be resolved by rehashing
- discuss properties of a good hash function
- indicate methods of choosing a hash function

Vocabulary: hash function, hash clash, collision, rehash function, clustering, linear probing, division method, midsquare method

Motivation:

- A postman delivering mail sorts the letters by putting them in the various mailboxes (this analogy works best if we think of mail being delivered to a large apartment building).

- What is the basis of the searching algorithm the postman uses? That is, how does the postman know which box each letter goes in?

  - The answer is, of course, that the letter has the address it goes to on it. The postman does not have to put the letter in order with respect to the other letters. Each letter "knows" where it belongs. Hashing is a technique which uses the information stored in a record to organize the table.

Aim: What is a hash-coded search?

Development:

1. In hashing, we use an array of "slots" into which the records to be sorted can be inserted. These slots are analogous to the individual mailboxes in the postman's situation. Each array slot has an address, and our object is to convert some of the information in the record to be sorted into an address. That way, we can look at the record and instantly tell where to find it in the array.

2. In mathematics, a correspondence between one set (here, the set of records), and another set (here, the set of indices for the array) can be studied using the notion of a function. It is assumed that students have some familiarity with this basic concept. A hashing function is a function that takes information from a record and returns a potential index (address) for the record in the array.

3. Not all hashing functions are equally useful. As an extreme example, we would not want to have a hashing function which returns only numbers from 1 to 10; chances are that we are sorting many more records, and this would not allow for room in the array for all of them. In fact, we ideally want the hashing function to spread out the records along the array, so that room is allowed for insertions and deletions.

- In particular, we want the ideal function to be one-to-one: If we start with different records, we want to get different hash values. Such a function is not easy to find. But there are ways to construct functions which will do a good job of spreading the records around.

  - One common way is to take a numerical field from the record, divide by 1000 (or some other large number), and take the remainder. If we used 1000, we would have 1000 different hashing values possible. If we are hashing only a few hundred records, we have a good chance to spread them out. This is called the division method of hashing.

  - Another method is to extract from a numerical field in the record some random number. Typically, the middle digits of the number in the record are removed to form a new number. This new number is then squared to get the hash value. This is known as the midsquare method.

4. But even with the best of hashing functions, we will sometimes run into the situation where two records are hashed into the same location. This is called a hash clash or collision. We need to resolve the clash by finding a new slot for the second record, from which we can retrieve it when necessary.

   - One way to do this is to place the record in the next available slot. This is called a linear probe. In searching for the presence of a record, we hash the key and look under the array entry given by the hash value. If it is occupied by a different record, we sequentially search the array. If the search is successful, we find the record. If it is not, we know to stop when we encounter a blank element of the array. This method is known as linear probing.

5. A linear probe will resolve a few clashes, but it tends to fill up continuous stretches of the array. If our original hash function is trying to spread the records out around the array, the linear probe is working against this by making them cluster.

   - If our hash function is like the midsquare function described above, it can be reapplied to get a new value. This new value is not likely to be near the old one. Thus, it should not lead to clumping up of records.

     - Reapplying a hash function to get a new value is called rehashing. Again, in searching for a record which may or may not be in the table, we continue rehashing until the search is successful or until the rehashed value gives a blank entry in the table.

   - But if our hash function involves a division method, applying the function twice will not yield a new result. In these cases, we may want to apply a new hashing function, with a different rule, to resolve clashes. The second function might be division by some other number. Or, if we are anticipating that it won't be used very much, it can be a more complicated function.

Application:

A. All 850 employees in Company ABC have 5-digit ID numbers. The ID number of each employee can be considered as a key value and converted to an array index which indicates where information about the employee is to be stored. The hash function H(key) = key MOD 1000 will store information using the last three digits of the ID number. Suppose 500 employees have been processed and the hash table has the following form:

	RECORDS	
	ID numbers (key)	INFO
0	52000	
1	83001	
2		
.	.	
.	.	
.	.	
621		
622	82622	
623	43623	
624		
625	71625	
626		
627		
628	84628	
629	63629	
630	71630	
631		
632	21632	
.	.	
.	.	
.	.	
998	23998	
999		

- If we now try to process ID number 37625, a hash collision occurs. The rehash function RH(K) = (K + 1) MOD 1000 implements the linear probe technique and stores ID number 37625 in location 626. Similarly ID number 16622 would be stored in location 624 by using the rehash function twice.

B. Why are the following not good rehash functions?

- RH(K) = (K + 500) MOD 1000
  - Each key can be placed in only two locations.
- RH(K) = (K + 5) MOD 1000
  - It is possible for only 200 out of a 1000 positions to be filled--for example, 2, 7, 12, 17, ..., 992, 997--and still no room remains for another record.

NOTE: Any rehash function of the form RH(K) = (K + A) MOD tablesize, where A and tablesize are relatively prime, produces successive rehash values that cover all indices of the table.

Summary:

1. What is hashing?

2. How can a hashing function minimize collisions?

3. What is a rehash function?

Exercise:

1. A school has 3000 students. Each student has a 9-digit social security number. It is desired to store the student records in an array by hashing.

   - How big should the array be?

   - Construct three different hashing functions which might be used.

   - How often would collisions occur? What would you do to resolve the collisions which do occur?

TOPIC 100

CHAINING

Performance Objectives - Students will be able to:

- cite problems relating to rehashing
- describe how collisions can be resolved by chaining
- explain a search and insert algorithm which implements chaining
- discuss advantages and disadvantages of chaining

Vocabulary: chaining, buckets

Motivation:

- In the postman situation of the last lesson, what does the postman do if there are too many letters to fit into one of the mailboxes?
- This real-life situation has a variety of answers (most of which do not have analogies in programming situations). But the question brings up an interesting difference between the postman's situation and the concept of a hash table as discussed in the previous lesson. This is the fact that many letters can fit into the same mailbox, while only one record fits into a slot in an array used for hashing. Sometimes we refer to a single space in the hashing array as a bucket.

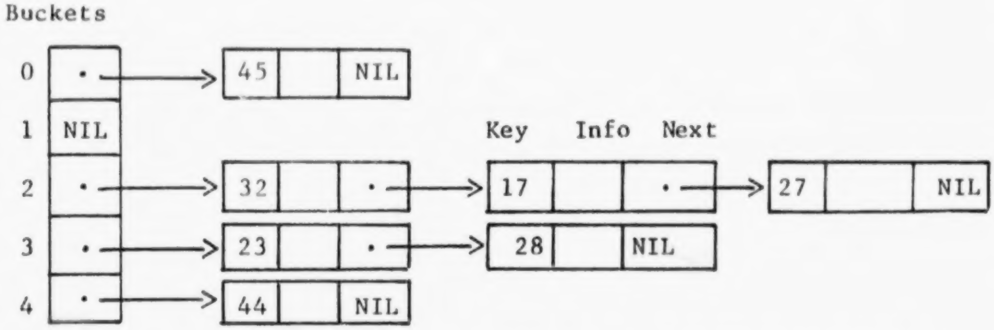
Aim: What do we do when the bucket overflows?

Development:

1. One common answer to this question is to not let the situation happen. If we resolve collisions by rehashing or by a linear probe, we keep no more than one record in each bucket. But linear probing splits up our table, and rehashing consumes valuable time.
    - Another way to prevent the situation is to use a bigger bucket. If the array has entries which hold, for example, three records, we can expect to reduce the number of collisions by a factor of 3. This can be implemented by having an N by 3 array.
2. This poses its own problem: Eventually we will be performing a large, time-consuming sequential search on each bucket. Even worse, some buckets may lie around empty (or nearly empty), while others get filled up.
3. This is the perfect sort of situation in which dynamic allocation can help. We can arrange to have flexible buckets which expand to accommodate as many records as possible, by dynamically allocating the storage space required.
    - Our original hash table can be a simple table of pointers. Initially, they are all NIL. Records are hashed to produce a hash value. The pointer at this entry in the table is then made to point to a dynamically allocated node, containing a copy of the record to be inserted, together with a new pointer.

- If another record is hashed to the same bucket (which is not a linked list), it is inserted in the list, again using dynamic allocation. The amount of room in the bucket is limited only by the amount of dynamically allocated memory available. The bucket can be maintained in sorted form, so that in retrieval we know quickly whether a given record is present. This method of resolving hash clashes is known as chaining.

4. Consider the hash function H(key) = key MOD 5, and the keys 32, 45, 23, 17, 44, 27, 28. An array of header nodes, called buckets, can be declared where bucket[I] points to the linked list of nodes whose hash value is I. The chaining method of resolving clashes would produce the following situation:



5. Chaining can be an ideal way to resolve collisions. But it does involve the additional overhead of pointers, a more complicated algorithm, and an environment which supports dynamic allocation. In particular, it may be complicated to implement if the hash table is stored externally.

Summary:

1. How does chaining resolve hash collisions?

Exercise:

1. Suppose you were using a hash table to store the names and birthdays of students in your class. The hash function being used assigns to each student the number of the month in which he/she was born. Thus, the hash table contains 12 elements. This is ridiculously small in real life, but for the purposes of this exercise, it will create many clashes to practice resolving.

   - What trouble would you quickly run into if you used linear probing or rehashing?

   - Suppose you used buckets which were large but fixed in size. How large would your buckets have to be to accommodate the entire class without trouble?

   - Suppose you used chaining to resolve the clashes. Draw a diagram showing the final organization of the hash table. Keep each bucket in alphabetical order.

Appendix

USING TURBO PASCAL

(Version 2.00B) on the I. B. M. P.C.

Turbo Pascal has many advantages over UCSD Pascal. Only one disk is necessary for loading, editing and compiling a program. Editing is much simpler and faster since you are always in the insert mode (unless you press a control key). Compilation time is almost instantaneous. Whereas, the operating system of UCSD would take one full week of instruction time to learn. Turbo can be learned easily after one day's practice. In short, once you are familiar with Turbo it is hard to return to UCSD.

Follow these steps:

1. Obtain a disk formatted with Turbo.
2. Insert the turbo disk, label side up in drive A (left drive or upper drive).
3. Close drive door.
4. Turn on your IBM.
5. When you are prompted to enter the date, either type in the date or a carriage return [CR].

Turbo Pascal Version 2.00B for the IBM PC

After loading your disk you will see:

```
A> turbo

-------------------------------------------------------------------------
Turbo Pascal System                                        Version 2.00B
                                                           PC-DOS

Copyright (c) 1983, 1984 by Borland, Inc.
-------------------------------------------------------------------------

Default display mode

Include error messages (Y/N)? -             (Type Y).
```

You will also see:

```
Loading A: Turbo.msg
```

The A indicates the drive being used: left or top drive if there are two drives; otherwise, the single drive becomes the default drive.

You will also see:

```
Logged Drive: A
Active Directory:  /
Workfile:
Mainfile:
Edit              Compile              Run              Save
Dir               Quit                 Compiler         Options
Text: 0 bytes
Free: 62635 bytes
```

>_ Type D to view the Directory

Dir Mask: Type A: or <CR>

This command causes all files currently on your disk in drive A to be listed. Type *.text to list files ending with .txt. Type Myfirst. pas to list Pascal files starting with Myfirst.

>_ Now type E(dit)

Workfile name: Type MyFirst <CR>

A message will appear: New File

You will see the status line:

```
Line 1     Column 1          Insert          Indent      A:  MyFirst.Pas
                        (you may change                  (work - (automatically
                        it to overwrite                   file    added
                        by typing Ctrl-V)                 name)   on)
```

All editing is done by first moving the cursor to the position into which you want to insert or delete.

You are now ready to type in your first program.

```
PROGRAM Heading (output);
BEGIN
  Writeln ('(************************************************************)');
  Writeln ('(*            Student's Name                                *)');
  Writeln ('(*            Student's Class                               *)');
  Writeln ('(*            Date                                          *)');
  Writeln ('(*            Teacher's Name                                *)');
  Writeln ('(*            Description of the Prog.                      *)');
  Writeln ('(* This prog. will create & store a heading                  *)');
  Writeln ('(* that I may reuse later                                   *)');
  Writeln ('(************************************************************)');
END.
```

```
  Ctrl-K, Ctrl-D or Ctrl-K,D
   S(ave)
   R(un)
```

If there is an error in the program, you may not be able to re-enter the editor. Therefore, always save your program before you run it. Alternatively, you could use C(ompile), S(ave) and R(un).

You will also see >_

(> is the prompt, - is the cursor)

If you now type E(dit), your last program will appear. You may also type D(ir Mask) or W(orkfile). Use W if you want to type your second program.

To Exit Turbo

Type Ctrl-K, Ctrl-D, or D, Q(uit) and you will see:

```
A>
A>_
```

You are now in DOS. You have to be in DOS to output to the printer, kill files or rename files. You will not return to Turbo until you type Turbo.

Using The Printer

Pressing the PrtSC and SHIFT keys simultaneously, will cause all information that appears on the screen to be printed (including extra material that is seen on the bottom of the screen). This command may have to be reapplied several times to print a long program that cannot fit on the screen at one time.

There are two ways to get a listing of your program on the printer.

1) You can change write statements from write (x) to write (LST, x).

2) Type:

```
copy heading.pas LPT1:
          or
copy heading.pas prn:
```

To direct output to the printer type:

`Type heading.pas > prn:`

NOTE: Yes, you type the word "Type".

Renaming or Killing Files

```
In DOS: A      Rename      First.pas      Heading.pas
   (FORMAT:    Ren         Old-Name       New-Name)
```

Now Type DIR: to see the change that has occurred.

You will also see the back-up copy, Myfirst.bak. This copy can be useful if you change a program, and afterwards decide that you prefer the previous version of the program.

Now Type: Erase   Myfirst.pas (the extension must always be used)

The DIR command will show that the file no longer exists.

Also note that the number of free bytes increases when files are erased.

Typing a Program

Your text is typed in from the keyboard, usually using the "insert" mode. Characters are displayed on the screen at the position of the cursor. The cursor moves freely when the user hits the arrow keys, the backspace key or a combination of keys described below.

(* The most useful options.)

CURSOR MOVEMENTS	KEYS USED
* 1) Left, Right, Up, Down	← → ↑ ↓ (arrow keys)
2) word left	Ctrl-A
3) word right	Ctrl-F
4) page up	Ctrl-R or 9 on Arithpad
5) page down	Ctrl-C or 3 on Arithpad
6) to top of page	Ctrl-Q, Ctrl-E
7) to botom of page	Ctrl-Q, Ctrl-X
* 8) top of file	Ctrl-Q, Ctrl-R
* 9) end of file	Ctrl-Q, Ctrl-C
*10) end of line	end 1 on Arithpad
*11) front of line	home 7 on Arithpad
INSERT & DELETE	
*10) Exchange (Erase and Replace)	Ctrl-V
*11) Delete Line	Ctrl-Y
*12) Insert Line	Ctrl-N
*13) Delete Char under cursor	Ctrl-G
*14) Delete char to left of cursor	Backspace
*15) Delete to end of line	Ctrl-Q, Ctrl-Y
*16) Mark beginning of Block	Ctrl-K, Ctrl-B
*17) Mark End of Block	Ctrl-K, Ctrl-K
*18) Copy Block	Ctrl-K, Ctrl-C

*19) Move Block	Ctrl-K, Ctrl-V
*20) Delete Block	Ctrl-K, Ctrl-Y
*21) Read Block from disk	Ctrl-K, Ctrl-R
*22) Write Block to disk	Ctrl-K, Ctrl-W

Strings: Strings may be used in Turbo Pascal, but are not part of Standard Pascal. Standard Pascal uses a packed array of characters to represent a string. To declare a word as a string, you must specify the length of the string; for example:

```
VAR
  Word:String[10];
or
TYPE
  Wordtype=String[10];
VAR
  Word:Wordtype;
```

In Turbo, Keyboard is abbreviated Kbd; printer is abbreviated LPT1 or PRN, and screen (or Console) is abbreviated CON.

Output may be sent either to the printer or to the screen when the following is inserted in your program:

```
VAR
  device: text;
  answer: char;
BEGIN
  Writeln ('Enter type of output.  Type S (screen) or P (printer) :');
  Readln (answer);
  IF answer = 'P' THEN'
    Assign (device, 'LPT1:')
  ELSE
    Assign (device, 'CON: ');
  Rewrite (device);
  Writeln (device, _________) (*modify your writeln statements so that the
                                 device is added to the beginning *)
 *Close (device);
 *(Not necessary since the compiler will automatically close the file.  If you
   are using the printer you may not get the last line of output without the
   close statement.)
```

We will now use Turbo to correct this Bug program:

```
PROGRAM Bug (Input - Output)
VAR
  BEGIN:  interger;
BEGIN;
  temp: = 13;
  Begin: = SQR (temp);
  Writeln "thirteen squared equals"; BEGIN
END;
```

After you finish typing program Bug, type Ctrl-Q, Ctrl-R to move the cursor to the top of the file so that you may edit it.

Move the cursor to

Program Bug (Input - Output)

Use:

- o INS or Ctrl-V to overwrite or exchange type, then
- o INS or Ctrl-V to get you back to insert.

NOTE: The status line will change.

Now move the cursor to type ;

Move the cursor down two lines and over to the B of BEGIN.

Type Ctrl-G five times, then type a new name, such as number.

Move the cursor over to the first r of interger. You can now use the delete key.

We have to insert a line so that temp can be declared. Either move the cursor to the end of the line past the ; and press CR or move the cursor to the end of the line, one space beyond the ;, and press Ctrl-N. With the cursor at the beginning of the blank line, type: temp: Integer; Move the cursor down one line and over to the ; following BEGIN; Press delete. Now go down two lines to BEGIN. You may press the backspace 3 times to delete G E B then press delete twice to delete IN; Now insert number.

Move the cursor down one line to writeln over to " then press del, and type ('; Go to " and delete ; then type '. With the cursor under ; press Ctrl-V, delete five times and type number. Go down one line to END and then left to END;. Press Ctrl-V, then., and go one space before END to backspace twice. Then press Ctrl-K, Ctrl-D, S, and R.

A pecularity of Turbo is that it cannot read a sequence of characters unless they are entered on separate lines. You may correct this deficiency with read(Kbd, CH) or through the use of a compiler directive.

To correct this defect use compiler directive {$B-} before your program heading. An example follows:

```
{$B-}

PROGRAM ReadvsReadln (Input, Output);
VAR
  ch:  char;
BEGIN
  writeln ('Enter a sentence that ends with a period ');
  repeat
    read (Ch); write (ch)
  UNTIL ch = '.';
END.
```

NOTE: A program heading is not necessary in Turbo Pascal, but should be included.

Sentences or segments of one program can be moved to another program or to within the same program. After typing a program, we might decide to rearrange lines without retyping. In the program to follow the writeln statement will be moved to a new position.

```
PROGRAM MultofTwo (Input, Output);
VAR
  N, Multiple: integer;
BEGIN
  FOR N := 1 TO 10 DO
    BEGIN
      Multiple := 2 * N;
      writeln (N)
    END;
  writeln (Multiple)
END.
```

This program attempts to print out all multiples of two from 2 * 1 = 2 to 2 * 10 = 20. Block commands can be used to correct the program by inserting writeln (multiple) within the loop. Move the cursor to the beginning of the line, press Ctrl-K, Ctrl-B (Mark begin of block). Move the cursor to the end of the line, press Ctrl-K, Ctrl-K; the color of the line will change so that you know your block has been marked.

Now move the cursor to the position where you want to move the block . Pressing Ctrl-K, and Ctrl-V will then move your block.

Copying a Program Into Another Program

You may copy a whole program from one program to another. Mark the beginning and end of the file you wish to copy. Put the cursor at the position in the new file where you want to insert and type Ctrl-K, Ctrl-R. The machine will prompt you "Read block from file:". When you type the file name, the machine will insert this entire program into its proper place.

To copy some segments of a program:

o Edit the program you wish to copy from;
o mark the beginning and end of the block and use Ctrl-K, and Ctrl-W.

The machine will prompt you to enter the file name that you are writing to. Name the file, even if the file has not been created yet. Then type Ctrl-K, and Ctrl-D, W and the name of the workfile.

Using Files with Turbo Pascal

Turbo Pascal uses the Assign statement to set up a correspondence between the file variable named in your program (Infile) and a file saved on your disk (for example, MorseCod. Dat).

1. Assign (Infile, 'MorseCod.Data');
2. Reset (Infile);
3. Read (Infile, ch);

NOTE: Reset opens the file Infile, so that characters can be read into it.

4. Close (Infile).

The close statement is not part of Standard Pascal, but should be used in Turbo, both in reading from a file and in writing to a file.

GLOSSARY

TERM	MEANING
ACTUAL PARAMETER	A variable or expression contained in a subprogram call that is actually passed to that procedure or function.
ADDRESS	A label for a specific memory location. The address can consist of a name, number, or symbol.
ALLOCATE	To set aside space in memory.
ALPHANUMERIC	Characters that are alphabetic or numeric-digits
ARGUMENT	A value or expression supplied to the parameters in a called procedure or invoked function.
ARITHMETIC OPERATOR	See operator, operand.
ARRAY	A collection of data items or elements that are all of the same data type. Each item or element is directly accessed by the index (see index).
ASCII	A standard code whose character set is composed of printable and control characters (American Standard Code for Information Interchange).
ASSEMBLER	A program that translates a high level language to machine language.
ASSIGNMENT OPERATOR	The Pascal symbol ":=" used in an assignment statement.
ASSIGNMENT STATEMENT	A statement that assigns a value to a variable (e.g. X := 3); equivalent to the Let statement in Basic.
BASE	The number of digits used in a number system (e.g. base 10 uses 10 digits).
BIG - O NOTATION	A measure of the relationship between the size of a file and the amount of time units needed to process it. Big O means the order of magnitude.
BINARY OPERATOR	An operator that requires two operands. (See arithmetic operator, relational operator.)
BINARY SEARCH	A searching technique that is used on an already ordered list. The list is repeatedly subdivided in half until we find our desired item or we have exhausted the list.

GLOSSARY

TERM	MEANING
BINARY TREE	A tree in which each node has at most two offspring. A complete binary tree of level N is a tree in which each node of level N is a leaf and each node which has a level less than N has non-empty left and right subtrees.
BINARY TREE SORT	A sort method in which items are put into a binary tree by comparing them with the contents of a node of the tree. When the item is less than the contents of the node a left branch is taken; otherwise, a right branch is taken. An inorder traversal of the completed tree produces a sorted list.
BIT	A Binary Digit (0 or 1) is used to represent information in the computer. A bit is the smallest unit of information used by the computer.
BOOLEAN EXPRESSION	An expression which takes on a value of true or false.
BOOLEAN OPERATOR	An operator that operates on boolean expressions to return a value of true or false.
BOTTOM-UP METHOD	A method of analyzing a problem proceeding from the specific to the general.
BUBBLE SORT	A sort which bubbles the smallest/largest element to the first position after the first pass is completed. During each pass every element is compared with its successor and is swapped if they are out of order.
BUCKET	A single space in the hash array that points to records which have the same hash value.
BUG	An error encountered in the compilation or execution of a program.
BYTE	A sequence of eight bits used to represent a value in storage.
CARDINALITY	The number of values contained in an ordinal type.

GLOSSARY

TERM	MEANING
CENTRAL PROCESSING UNIT (CPU)	The "brain" or hardware of the computer that interprets and executes instructions, and communicates with input/output and storage devices. It comprises the control unit, arithmetic logic unit and registers.
CHAINING	A method for resolving hash collisions in which each bucket points to a linked list or chain of records.
CHARACTER SET	A set of machine representable characters encoded according to a specific coding system.
CIRCULAR LINKED LIST	A linked list in which the last node has as its next address the location of the first node.
CLUSTERING	A situation where keys that initially hash to different values have collisions after rehashing.
CODE	In general, the instructions that are processed by the computer. (See source code and object code).
COMMENT	See documentation.
COMPATIBLE	Variables or expressions are type compatible if they represent values of the same underlying type.
COMPILER	A computer program that translates a source program, written by a user or programmer in a high level programming language, into machine language (object code).
COMPILER DIRECTIVE	A command to the compiler to interrupt the normal compilation process. A compiler directive begins with a bracket immediately followed by a $. It then contains a letter or letter and number, and possibly a + or - ; then a concluding bracket.
COMPOUND STATEMENT	A group of program statements sandwiched between the **Pascal** reserved words BEGIN and END, that may be treated as a single program statement.
COMPUTER	A programmable electronic device that is used to process and store data.

GLOSSARY

TERM	MEANING
CONCATENATE	Combining of 2 or more groups of data (usually files and strings) by putting them in sequence. (e.g. the concatenating of "MAMA" and "MIA" results in "MAMAMIA").
CONDITION	A boolean expression that is the basis of comparison at a conditional program branch.
CONSTANT	An identifier whose value cannot be changed (e.g. pi=3.14159 or BLANK=' ') that is declared in the constant definition section of a program.
CONTROL CHARACTER	A byte of information that does not correspond to any in the standard character set. A carriage return is the most frequently used control character. Control characters have ASCII numeric equivalents of 0 through 31.
CONTROL STRUCTURE	A statement that controls the execution of an action and determines the order of processing.
CRASH	The sudden halting of computer execution due to system malfunction or run time errors.
CRT	Cathode Ray Tube is a television-like electronic tube, upon which visual information may be displayed (used as a computer video terminal).
DANGLING POINTER	A pointer that contains a valid address, but where the contents at that address have been freed (e.g. by dispose (P)).
DECLARATION PART	Section of a program or program block that precedes program statements and declares identifiers to be used in the program statements.
DEFAULT VALUE	A value that is assumed by the system or compiler when no value is assigned by the program or the user.
DEGREE OF A NODE	The number of sons of a node in a tree.
DELIMITER	A word (begin,end)(repeat,until) or symbol ([ ]) used to indicate the boundaries of a group of statements.
DIRECT ACCESS	The process of retrieving or storing elements in a data structure, in time that is independent of the location of the elements. (Also called random access.)

GLOSSARY

TERM	MEANING
DISPOSE	A procedure which frees storage space. Dispose is a reserved word in Pascal.
DIVISION METHOD	A hashing method that divides a key value by a constant, and uses the remainder as the hash value.
DOCUMENTATION	A detailed written description of specifications, design, code and comments which make a program easier to read, understand and modify.
DOUBLY LINKED LIST	A linked list in which every node contains two pointers, one to the next node and one to the previous node.
DRIVER PROGRAM	A program developed to test subprograms.
DYNAMIC ALLOCATION	Creation of storage space in memory during run time.
DYNAMIC DATA STRUCTURE	A data structure that may expand or contract during the running of the program (e.g. linked lists or binary trees implemented with pointer variables).
EBCDIC	Extended Binary Coded Decimal Interchange Code. A system used by IBM to represent characters.
EDITOR	An interactive program of the operating system, that allows for the creation and modification of program text.
EFFICIENCY	A relative measure of a program's usage of computer resources. (e.g. fast speed and minimum allocations of memory improve program efficiency.)
EMPTY STATEMENT	A syntax implying no action; usually marked by a semicolon (e.g. used in a case label list).
ERROR CHECK	The checking of input for incorrect data that would cause a program crash or give incorrect results.
ERRORS	Compiler error - a syntactical error that is found when the program is being compiled or translated (it must be fixed before the program is run). Output error - An error in output caused by a faulty algorithm or incorrect data.

GLOSSARY

TERM	MEANING
EXECUTION TIME	The time required to execute the program instructions (also known as run time).
EXTENSION	A nonstandard addition to **Pascal**. UCSD Pascal and Turbo Pascal offer many text string manipulations which are not part of standard **Pascal**.
EXTERNAL SORT	A sort in which all the records being sorted are not stored in the main memory.
FIELD IDENTIFIER	The name of a component in a record.
FIELD SELECTOR	A method to access a field of a record by first naming the record, followed by a period and the name of the field.
FIELDWIDTH	A positive integer that follows the optional colon in a writeln or write in statement. This integer gives the minimum **fieldwidth** for a value that is to be printed.
FIFO	First In First Out. A term used for a queue. The term indicates the order in which items are accessed from a queue.
FILE	A data structure consisting of a sequence of components that are all of the same data type; a collection of related data, usually residing in secondary storage, that is referenced by a single name.
FILE BUFFER VARIABLE	A variable of the same type as the components of the file with which it is associated; used as a "window" through which we can read or write file components.
FILE UPDATE	The changing of some part or component of a file.
FIXED POINT NOTATION	A method of writing real numbers in which no exponents are used. **Pascal** allows real numbers to be printed without exponents by using fieldwidth specification.
FLAG	Usually a boolean variable that changes value to represent a change in an event (e.g. found := false;) (see sentinel).

GLOSSARY

TERM	MEANING
FLOATING POINT NOTATION	A method of expressing a real number as a factor of some power of 10. In floating point notation, a real number is represented with a decimal point, and the letter E followed by an appropriate power of 10 (e.g. 43.67 is equivalent to 4.367E01 which means 4.367 times 10 to the 1st power.
FORMAL PARAMETER	A variable declared and used in a procedure or function heading.
FORWARD REFERENCE	A forward reference is a statement used so that a function or a procedure can be referenced before it has been declared. The reference informs the compiler that the declaration for the function procedure will come later in the program.
FUNCTION	A subprogram that when invoked returns one value.
GLOBAL IDENTIFIER	An identifier declared in the main program.
HARDWARE	Any physical equipment used with the computer system (that is, the computer and any peripheral devices).
HASH FUNCTION	A function that takes information from a record and returns a potential address for that record.
HEADER NODE	Dummy node at the beginning of a list used to hold information about the list.
HEADING	The first line of a program or subprogram (if any).
HEXADECIMAL	Base 16.
HIGH LEVEL LANGUAGE	A programming language whose instructions are easily understandable by humans.
HUFFMAN ALGORITHM	A method of minimizing the number of bits needed to store messages in the computer's memory.
INDEX (SUBSCRIPT)	An ordinal number that specifies the location of an element.
INDIRECT RECURSION	(Mutual recursion) One subprogram will call a second subprogram which will eventually call the first subprogram back.

GLOSSARY

TERM	MEANING
INFIX NOTATION	The notation in which each binary operator is between its operands in an algebraic expression (e.g. A+B*C).
INITIALIZE	The process of giving a starting value to a variable or register (usually prior to processing). Some systems might automatically initialize variables and registers.
INORDER TRAVERSAL	A recursive traversal of a binary tree in which the root is visited in-between the left subtree and the right subtree.
INPUT	Data that is entered into a system for processing.
INSERTION SORT	A sorting technique that inserts each succeeding element (after the first), into an existing sorted list in such a manner that the resulting list remains sorted.
INTERNAL SORT	A sort in which all the records being sorted are stored in the main memory.
INTERACTIVE FILE	A file type in UCSD **Pascal** for files of character that are read or printed as they are created. Input/output and keyboard are interactive files.
INTERPRETER	A program that translates source code (high level language) directly into machine language. Each statement of the source code is translated and immediately executed.
INVOKE (CALL)	To execute a function or procedure, by the appearance of the function or procedure name (identifier) in a program statement.
ITERATION	Repetition of program statements. Sometimes referred to as program looping.
JOSEPHUS PROBLEM	A problem to determine the remaining item in a circular grouping, if each K^{th} successive item is eliminated.
KEY	Field of a record which is used to sort or search records.

GLOSSARY

TERM	MEANING
KEYBOARD FILE	A file, unique to some **Pascal** systems, which is used with a read statement to read a character from the keyboard without echoing it on the screen (USCD: READ(KEYBOARD,X); TURBO:READ(KBD,X)).
LABEL	A positive integer used in a program to redirect the flow of control from one portion of the program to another. Labels are used with goto statements.
LEAF	A node of a tree which has no children.
LEVEL	The root of a tree has a level 0, and the level of any node is one more than its parent. The maximum level of all nodes is the level of the tree.
LIBRARY	A collection of nonstandard procedures that are available to the programmer.
LIFO	Last In First Out. A name used for a stack. The term indicates that the last item pushed onto a stack will be the first item that is popped.
LINEAR LINKED LIST	A data structure in which each element contains within itself the address of the next element.
LINEAR PROBE	A method to resolve a hash clash, whereby a second record that has clashed with another is moved to the next available location using a sequential search.
LITERAL	Data that is written between single quotation marks. Literals are printed exactly as they appear between the quotation marks.
LOCAL IDENTIFIER	An identifier declared or redefined in a block (e.g. procedure or function) that is said to be local or restricted to that portion of the program.
LOGICAL OPERATOR	See boolean operator.
LOOP	A control structure that allows for the repetition of a series of statements, based on a certain condition test. (See iteration.)

GLOSSARY

TERM	MEANING
LOOP CONTROL VARIABLE	A variable used to control the number of times the body of the loop is executed.
MACHINE LANGUAGE	A language made up of a series of 0's and 1's that is understood by the computer.
MATRIX	A 2-dimensional array.
MAXINT	Predefined identifier whose value is set to the largest integer number, that can be represented by the computer (e.g. 32,767).
MERGE SORT	A sort which combines previously sorted lists into one sorted list.
MID-SQUARE METHOD	A hashing technique, where the middle digits of a number are squared to get a new index value.
MODULARITY	A property which describes programs that have been subdivided into independent subprograms (modules) each of which can communicate with the rest of the program.
NESTING	A statement or series of statements that is identical in form to a larger unit in which it is contained.
NEW	A procedure that creates the space for a node. "NEW" is a reserved word in Pascal.
NIL	A value that can be assigned to a pointer variable, to indicate that the pointer points no where.
NODE	Each element of a linked list.
NONLOCAL IDENTIFIER	An identifier that is not local to a block of the program.
OBJECT CODE	Machine language equivalent of a source code. The object code (machine code) results from compiling the source code.
OCTAL	Base 8.

GLOSSARY

TERM	MEANING
OPERAND/OPERATOR	An operator is a symbol that indicates the type of operation to be performed. Operands are the expressions which are operated upon to produce a value (e.g. in A + B, A and B are operands and + is an operator).
OPERATING SYSTEM	A series of control programs that directs the use of computing tasks such as input/output, compilation, and scheduling.
OPERATOR PRECEDENCE	The order in which operations are to be performed in an expression.
ORDINAL TYPE	A set of distinct values that are ordered, such that every value has a unique predecessor and successor, except for the first and last. Integer, char, and boolean are ordinal types; real is not.
OUTPUT	Data that has been processed by the computer.
OUTPUT ERROR	Output error is an error in output caused by a faulty algorithm or incorrect data.
OVERFLOW	A condition where the results of an operation are too large to fit into an existing structure.
PACKED ARRAY	An array in which more than one character is stored in each memory cell, so that less storage space is used.
PARAMETER	A variable created within a subprogram into which values can be passed, or from which values may be returned.
PASS BY REFERENCE (ADDRESS)	A parameter that is passed by reference (or address) results in the actual and formal parameters sharing the same memory location; if one is changed the other must also be changed. It is distinguished from a value parameter in formal parameter lists by the keyword VAR. Used to input/output values in a subprogram.
PASS BY VALUE	A parameter that is passed by value has a copy of the parameter made. The procedure or function called may change the value of the copy, but can never change the actual parameter. Used to input values to a subprogram.

GLOSSARY

TERM	MEANING
POINTER	A variable that holds the address of a node.
POP	An operation which removes the top element of a stack.
POSTFIX NOTATION	The notation in which each operator is located after its operands (e.g. AB+).
POSTORDER TRAVERSAL	A recursive traversal of a binary tree in which the root is visited first, then the left subtree, and then the right subtree.
PREFIX NOTATION	The notation in which each operator precedes its operands (e.g. +AB).
PREORDER TRAVERSAL	A recursive traversal of a binary tree in which the root is visited first, then the left subtree, and then the right subtree.
PRIMARY KEY	First key on which we sort.
PROCEDURE	A subprogram that is executed when called.
PROCEDURE CALL (INVOCATION)	Activating the procedure by using the procedure's name along with any of its parameters.
PROGRAM CORRECTNESS	A program is correct if it meets all of its specifications.
PROGRAM VERIFICATION	A theoretical branch of computers science that attempts to prove a program is correct (as we prove theorems are correct in geometry).
PROMPT	A message displayed to indicate to the user the type of action that must be taken to continue processing the data.
PSEUDO-CODE	Description of an algorithm in English-like statements that follow the general structure of the program.
PUSH	An operation which adds an element to the top of a stack.
QUADRATIC SORTING TIME	Algorithm with a time of order N-squared, or O(N square).

GLOSSARY

TERM	MEANING
QUEUE	A linear structure in which elements are entered at one end (rear) and removed from the other (front). (See FIFO)
RADIAN	A unit of measure for angles in which 2 radians = 360 degrees. One radian is approximately 57.3 degrees.
RAM	Random Access Memory. The part of the computers memory that can be accessed or altered as needed by the program. It is used for storing programs and data.
RANDOM ACCESS	A structure in which the time to retrieve data is not affected by the order of the data in the structure.
RANGE	See Real.
REAL	A number that has a whole and fractional part. Precision: The maximum number of significant digits. Range: The smallest and largest allowable values.
RECORD	A data structure that can store different types of data in fields.
RECURSION	A technique in which a procedure or function calls itself.
REHASH FUNCTION	A function that transforms an index into a second index.
RELATIONAL OPERATOR	A symbol that operates on two compatible expressions to yield a value of true or false (e.g. less than, greater than).
RESERVED WORD	A predefined identifier that has a specific meaning and whose meaning cannot be changed.
ROBUSTNESS	A program is said to be robust if it is resistant to user errors.
ROOT	Top or base node in a binary tree.

GLOSSARY

TERM	MEANING
ROM	Read Only Memory. The part of the computer memory that contains prewired functions. This part of the memory cannot be altered by the programmers instructions.
SCALAR TYPE	A set of distinct values (constants) that are ordered.
SCOPE	The range or area within a program in which an identifier is known or has meaning .
SELECTION SORT	A sort in which successive elements are selected from a list and put in their proper position.
SENTINEL	A special value used to mark the end of input (e.g. -1, if entered data is positive). (See Flag)
SEQUENTIAL SEARCH	Locating a particular element in a data structure by checking each record in the given order.
SET	A structured data type composed of a collection of distinct elements according to some common property or rule.
SHELL SORT	A sort that repeatedly subdivides a file by diminishing increments, and sorts the smaller subfiles.
SIDE EFFECT	The change of a global variable from within a subprogram.
SIMPLE TYPE	One of the standard types (real, boolean, integer, char), or any user defined ordinal or subrange type.
SOFTWARE	Any computer program that is available to the computer.
SOURCE CODE	Source program. A program written in symbolic programming language by the programmer. The source program must be translated or compiled before it is executed.
SPARSE MATRIX	A 2-dimensional array whose elements are mostly zeroes. A sparse matrix is most efficiently implemented with linked lists.

GLOSSARY

TERM	MEANING
SPECIFICATION	Requirements and assumptions about the problem to be solved.
STACK	A data structure which allows access only at one end, called the top. (See LIFO)
STACK OVERFLOW	An error caused by trying to add an item (push) to a full stack.
STACK UNDERFLOW	An error caused by trying to remove an item (pop) from an empty stack.
STANDARD FUNCTION/PROCEDURE	Subprograms that are predefined in the computer system (e.g. SQR,SQRT,).
STANDARD PASCAL	The official **Pascal** language that was standardized by Niklaus Wirth in 1968.
STATEMENT	An instruction in a programming language.
STEPWISE REFINEMENT	A method of programming in which an abstract algorithm is stated, then successively refined and restated until it can be implemented.
STRING	A collection of characters interpreted as a single data item (sometimes known as a packed array of character values).
STRUCTURED TYPE	One of the standard types of data structures: array, set, record or file.
STUB PROGRAM	A working shell of a program, which allows the basic ideas behind the program to be tested.
SUBRANGE	A subrange type is defined by two constants of type integer, char, or scalar, and a sequence of values in between the two constants.
SUBSET	Set A is a subset of subset B, if each element in A is also an element of set B.
SYNTAX	Rules governing the formation of program statements.
SYSTEM DEPENDENT	A feature or value that can vary with the system being used.

GLOSSARY

TERM	MEANING
SYNTAX DIAGRAM	A diagram depicting the syntax of a Pascal statement.
SUBPROGRAM	A smaller program used to do a specific task within a larger program (e.g. function or procedure).
TABLE	A structure that holds a set of records (e.g. an array, a file, a linked list).
TAGFIELD	The case selector variable in variant records.
TEXT FILE	A file of characters that is divided into lines, as in a text.
TEXT PROCESSING	The processing of text; working with characters.
TIME SHARING	A term used when several users share the "CPU" at the same time. This is usually done with the use of terminals.
TOP DOWN METHOD	A method of breaking down a program from its most general form (top) into more detailed modules.
TOWERS OF HANOI	A problem that involves moving disks from one of three pegs to another by specific rules.
TRACE	Charting the logical flow of a program to determine the values of variables after each instruction.
TRAILER POINTER	If we use two pointers to traverse a linked list and one node is always one behind the other, the second pointer is called the trailer pointer.
TRAVERSE A TREE	To pass through each node of a tree.
TREE	A data structure in which a root can have an offspring, each of which can also be roots and have offsprings.
TRUNCATE	To "chop off" the decimal portion of a number (e.g. 7.85 truncated is 7).
UNDERFLOW	A condition where the result of a calculation is too small to be represented on the computer (opposite of overflow).

GLOSSARY

TERM	MEANING
UCSD PASCAL	A version of Pascal developed by the University of California at San Diego.
UNARY OPERATOR	An operator reauiring only one operand, such as the logical operator "NOT".
USER	The person or group for whom a computer program is written.
USER-DEFINED SCALAR	A scalar type that is defined by the user, e.g. TYPE color = (red, green blue); also known as user defined ordinal type.
VALUE PARAMETER	A formal parameter that is a local variable of a procedure or function, but whose starting value is passed as an argument to the subprogram.
VARIABLE PARAMETER	A formal parameter that is replaced by an actual parameter in a call to a procedure or function.
WORD	One or more bytes grouped together into a unit.

EVALUATION QUESTIONNAIRE

New York City Board of Education
Teacher's Guide to Advanced Placement Computer Science With Pascal - Volume II

This is an experimental bulletin in computer education. It is a pioneering effort to standardize computer instruction on a citywide basis. Your suggestions for improvement are most welcome.

1. How many classes in your school pursued this course of study? ____________

2. What were the grade levels of the students taking this course? ____________

3. Is the format of the bulletin useful? ____________

4. Is the bulletin well organized? ____________

5. Are the goals of the computer literacy course clearly stated?

   ____

6. Are the activities appropriate for the intended grade levels?

   ____

7. Is there sufficient emphasis on enduring computer concepts, such as good documentation, structured programming, modular design, problem-solving skills, and so forth? ____________

   ____

8. Did the learning activities enable students to achieve the performance objectives? ____________

   ____

9. Was there a significant variety of activities in the learning strategies?

   ____

   ____

10. What changes (if any) would you make in the learning activities presented in the bulletin? ____________

    ____

11. What additions would you make to the scope of the course content?

    ____

    ____

12. What topics in the course of study would you delete or change?

13. Was the interest level of the materials high among the students?

14. What changes in sequence would you recommend?

15. What difficulty (if any) did you encounter in implementing this course?

16. What do you consider to be the overall strengths of the bulletin? (If any)

17. What do you consider to be the overall weaknesses of the bulletin? (If any)

18. What changes would you suggest to improve the quality of the bulletin? (If any)

19. Please make any additional comments you feel are appropriate.

Return to: Mr. Marc Licht, Director
Computer and Information Sciences Unit
131 Livingston Street - Room 409
Brooklyn, N.Y. 11201